Titre: Title:	Outil d'analyse de performance d'applications par objets répartis
Auteur: Author:	Daniel Grigoras
Date:	2004
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Grigoras, D. (2004). Outil d'analyse de performance d'applications par objets répartis [Master's thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/7253/

UNIVERSITÉ DE MONTRÉAL

OUTILS D'ANALYSE DE PERFORMANCE D'APPLICATIONS PAR OBJETS RÉPARTIS

DANIEL GRIGORAS
DÉPARTEMENT DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
JANVIER 2004

Your file Votre référence
ISBN: 0-612-89207-7
Our file Notre référence
ISBN: 0-612-89207-7

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

OUTILS D'ANALYSE DE PERFORMANCE D'APPLICATIONS PAR OBJETS RÉPARTIS

présenté par: GRIGORAS Daniel

en vue de l'obtention du diplôme de: Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen constitué de:

M. BOIS Guy, Ph.D., président

M. DAGENAIS Michel, Ph.D., membre et directeur de recherche

M. ROY Robert, Ph.D., membre

REMERCIEMENTS

Je remercie tout d'abord mon directeur de recherche Michel Dagenais pour m'avoir accordé sa confiance durant toute la période d'études et de m'avoir fait profiter de sa grande expertise et de son expérience. Ses conseils et suggestions concernant les directions et les points ciblés de recherche ont rendu ce travail possible.

Je remercie les développeurs du projet Mono qui partagent généreusement les résultats de leur travail, dans un vrai esprit *source libre*.

Enfin, je remercie mes amis et ma famille qui m'ont toujours soutenu pendant tout ce temps.

RÉSUMÉ

La motivation pour construire et utiliser des systèmes répartis vient d'un désir de partager des ressources et de la nécessité d'avoir des systèmes redondants pour tolérer les pannes. Les applications réparties séparent les processus vivant dans un environnement hétérogène, de sorte qu'aucun processus ne s'arrête pour 'attendre' les autres. C'est une des conditions pour concevoir des systèmes performants client/serveur à grande échelle.

.Net Remoting simplifie l'architecture d'accès d'objet distant et permet d'étendre l'interaction afin d'utiliser n'importe quel protocole à travers de toutes les couches de transport. L'infrastructure .Net Remoting cache les aspects de bas niveau des composants fondamentaux, tous configurables, ce qui facilite le travail de développeurs. La pénalité au temps introduite par l'intergiciel, nécessaire pour soutenir des interactions entre les clients et le serveur, accroît le temps de réponse total. Celui-ci inclut les *temps de service des ressources* qui décrivent comment le serveur se comporte à l'intérieur, fournissant l'explication technique pour les caractéristiques d'exécution observées de l'extérieur.

Le temps de réponse est la plus importante des exigences du client. C'est la raison pour laquelle l'analyse des performances doit prendre sa place dans toutes les étapes du développement. Tôt ou tard, quelqu'un examinera d'une manière critique le système. Le but de ce mémoire est de mesurer et d'analyser les composants du temps de réponse pour les appels .Net Remoting.

L'outil d'analyse des performances que nous proposons donne une mesure du temps d'exécution de tous les événements passés pendant le dialogue entre le client et l'objet serveur. Il aidera les développeurs d'applications réparties à trouver plus rapidement les responsables des goulots d'étranglement et à conséquemment optimiser la performance.

Mots-clés: .Net Remoting, CLI, interactions client/server.

ABSTRACT

The motivation for constructing and using distributed systems arises from a desire to share resources and a need to build redundant systems. Distributed applications are separate processes living in a heterogeneous environment, so that no one process ever needs to stop processing to wait for the other(s), one of the key starting point for designing high–performance enterprise client/server systems.

.Net Remoting simplifies the remote object access architecture, giving also the option to extend remoting to include virtually any protocol on any transportation layer the developer comes across. .Net Remoting infrastructure hides many low-level aspects of underlying components, all of them configurable, which make easier developers' work.

The overhead introduced by this middleware, needed to support interactions between clients and servers, adds to the response time. It includes *resource service times*, describing how the server behaves internally, providing the technical explanation for the externally observed performance characteristics. Response time is at the top of client requirements, and performance analysis is important in all stages of the development; sooner or later, someone will monitor the system performance.

The goal of this project is to understand and analyze the components of response time for .Net Remoting calls. The performance analysis tool we propose measures the execution time of all the events occuring during the client-server dialogue.
It will help distributed applications developers to quickly get to the root of the bottlenecks and to accurately monitor performance improvements.

Keywords: .Net Remoting, CLI, client/server interactions

TABLE DES MATIÈRES

REMERCIEMENTS iv

RÉSUMÉ v

ABSTRACT vi

TABLE DES MATIÈRES vii

LISTE DES TABLEAUX ix

LISTE DES FIGURES x

LISTE DES NOTATIONS ET DES SYMBOLES xii

LISTE DES ANNEXES xiv

CHAPITRE 1 INTRODUCTION 1

CHAPITRE 2 REVUE DE LA LITTÉRATURE 5

2.1 Notions introductives 5

2.2 Java RMI 13

2.2.1 ProActive - IC2D

Interactive Control & Debug for Distribution 16

2.2.2 JaViz 18

2.3 CORBA 21

2.3.1 Le traçage d'une application répartie sur CORBA 25

2.4 NetLogger Toolkit 27

2.5 Les outils commerciaux et les travaux apparentés 30

2.6 Les limites des approches proposées 33

CHAPITRE 3 CONCEPTS DE BASE 37
3.1 Notions introductives CLI 37
3.1.1 La sécurité d'application et l'intégration inter-langage . . . 37
3.1.2 Les métadonnées 43
3.1.3 L'assemblage 44
3.1.4 Les fils d'exécution 47
3.1.5 JIT 48
3.1.6 Gestion de la mémoire 50
3.2 .Net Remoting 54
3.2.1 Les messages et les composants d'objet mandataire 54
3.2.2 Les récepteurs IMessageSink 58
3.2.3 Les canaux 60
3.2.4 Les récepteurs formateurs 61

CHAPITRE 4 LA PROPOSITION EXPÉRIMENTALE 65

CHAPITRE 5 LES RÉSULTATS 75

CHAPITRE 6 CONCLUSION 96

RÉFÉRENCES 98

ANNEXES 104

LISTE DES TABLEAUX

Tableau 3.1 Comparaison entre les fils d'exécution CLI 48

Tableau 4.1 Le surcoût d'instrumentation 66

Tableau 4.2 Nombre de méthodes pour appel local et en réseau 67

Tableau 5.1 L'appel numéro 1 86

Tableau 5.2 L'appel numéro 2 86

LISTE DES FIGURES

Figure 1.1	Vue d'ensemble: application répartie	2
Figure 2.1	L'intergiciel et le contexte environnant	10
Figure 2.2	La pénalité d'un accès réseau pour des arguments de types primitifs dans le cas de différents protocoles sous-jacents [34]	12
Figure 2.3	L'architecture RMI	13
Figure 2.4	IC2D: vue du journal pour un échange de messages	18
Figure 2.5	JaViz: détailles journalisés pour l'appel d'une méthode	20
Figure 2.6	CORBA esperanto	22
Figure 2.7	CORBA: les composants	22
Figure 2.8	L'analyse de traçage d'interactions dans l'application CORBA	26
Figure 2.9	NetLogger: visualisation d'interactions entre processus	29
Figure 3.1	Le système global	38
Figure 3.2	L' hiérarchie des types	39
Figure 3.3	L'espace d'adressage, domaines d'application, assemblages et modules	41
Figure 3.4	Une vue d'un processus avec CLI et deux domaines d'application. À chaque Type corespond un ensemble de métadonnées [65]	46
Figure 3.5	Les composants d'un domaine d'application	47
Figure 3.6	Modèle d'éxecution	49
Figure 3.7	L'infrastructure CLI répartie	52
Figure 3.8	*.Net Remoting* - vue d'ensemble	55
Figure 3.9	Les types de messages	55
Figure 3.10	La vue du développeur	56
Figure 3.11	*.Net Remoting* - vue en détail [64]	63

Figure 4.1	Les sous-composants pour l'application client. Le diagramme montre bien le trop grand nombre d'éléments à ce niveau de détail.	69
Figure 4.2	Cycle caractéristique d'un appel distant	70
Figure 4.3	La traduction d'événements primaires en événements abstraites	71
Figure 4.4	L'interaction client - serveur	73
Figure 5.1	Appel 1 en local	76
Figure 5.2	Appel 2 en local	76
Figure 5.3	Appel 3 en local	77
Figure 5.4	Appel 1 en réseau	81
Figure 5.5	Appel 2 en réseau et les détails de fin du dialogue	82
Figure 5.6	Le temps de réponse - en local et en réseau	84
Figure 5.7	Appel 2 en local - Test 1	89
Figure 5.8	Appel 2 en local - Test 2	90
Figure 5.9	Appel 2 en local - Test 3	91
Figure 5.10	Appel 2 en local - Test 1	92
Figure 5.11	Appel 2 en réseau - Test 1	92
Figure 5.12	Appel 1 en local - Test 3	93
Figure 5.13	Appel 2 en local - Test 3	93
Figure 5.14	Les appels locaux - Tests 1,2,3	94

LISTE DES NOTATIONS ET DES SYMBOLES

- CLI Common Langage Infrastructure
- ISO International Organization for Standardization
- TR Tehnical Report
- TSC Time-Stamp Counter
- AOT Ahead Of Time
- ECMA Europeean Computer Manufactured Associations
- OMG Object Management Group
- NTP Network Time Protocol
- CORBA Common Object Request Broker Architecture
- RMI Remote Method Invocation
- DCOM Distributed Component Object Model
- SOAP Simple Object Access Protocol
- IIOP Internet Inter-ORB Protocol
- GIOP General Inter-ORB Protocol
- IDL Interface Definition language
- RPC Remote Procedure Call
- JRMP Java Remote Method Protocol
- CDR Common Data Representation

- XDR eXternal Data Representation
- ORB Object Request Broker
- JVM Java Virtual Machine
- UCT Unité Centrale de Traitement

LISTE DES ANNEXES

ANNEXE I 104

ANNEXE II 106

ANNEXE III 125

CHAPITRE 1

INTRODUCTION

La première étape vers la simplification de la mise en œuvre des applications réparties est l'utilisation d'intergiciel comme base des objets répartis; l'intergiciel cache la complexité de la communication. Les intergiciels les plus courants suivent le modèle de programmation orienté objet et, par conséquent, une application répartie est un groupe d'objets distants interconnectés entre eux.
Les modèles orienté objet ont montré leurs limites: tâches qui doivent être gérées manuellement, évolutions/modifications difficiles, le déploiement d'applications à gérer [66].

C'est pourquoi, certains industriels, comme Microsoft [21], SUN [57] et l'Object Management Group (OMG [23]) ont adopté des modèles d'architecture par composants, pour simplifier le développement d'applications de plus en plus complexes. Une application répartie (fig. 1.1) est une application qui, selon les principes de l'architecture client-serveur, peut tourner de façon transparente sur plusieurs ordinateurs reliés en réseau.
Parmi les raisons pour cette alternative à la programmation orientée composant on peut mentionner:

- la construction des applications par l'assemblage des entités existantes (composants);
- la définition d'une architecture logicielle (connecteurs et schémas de connexions entre les composants);
- le formalisme pour décrire les interactions entre les composants;
- le formalisme pour décrire le déploiement des composants.

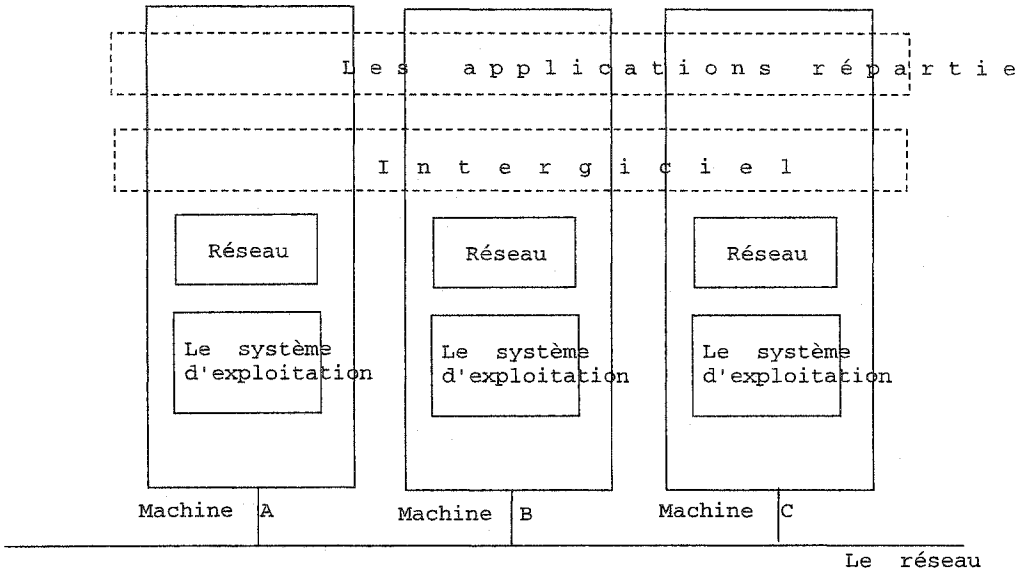

Figure 1.1 Vue d'ensemble: application répartie

Le développement de systèmes répartis est simplifié par l'existance d'infrastructures *intergicielles*, qui:

- traitent des problèmes d'hétérogénéité et d'interopérabilité, de manière transparente pour les applications, et
- fournissent des services réutilisables qui résolvent des problèmes de gestion de la distribution, souvent rencontrés dans la pratique.

L'analyse de performance d'un intergiciel est une tâche complexe. Elle caractérise les differentes étapes du cycle de vie d'une application ainsi que les phases de conception, de migration à de nouvelles plate-formes et d'optimisation pour une plate-forme particulière.

Les applications client-serveur/réparties ont évolué à partir de CORBA, Java RMI et DCOM jusqu'aux plus récents, les services Web.

Chacune de ces technologies présente ses propres avantages et désavantages en

matière de fiabilité, de tolérance aux pannes, d'adaptabilité, de sécurité et du coût caché de maintenance.

En janvier 2002 (après 18 mois de la mise à disposition de la version beta) l'architecture .NET de Microsoft est sortie. Cette architecture s'appuie sur des standards déposés à l'ECMA et acceptés par l'ISO [15] (décembre 2002): ISO/IEC 23270 (C#), ISO/IEC 23271 (CLI) et ISO/IEC 23272 (CLI TR).

Le projet Mono [68] est sorti en juillet 2001 ayant pour but d'implanter la nouvelle architecture .NET sur Linux.

Mono s'avère être un environnement idéal pour l'analyse des performances, son code source est librement modifiable, ce qui permet d'adapter certains composants du compilateur ou des librairies pour une étude approfondie.

Comme état de l'art en matière de communications entre les applications réparties, on trouve dans la nouvelle architecture .NET/Mono la librairie *Remoting*. L'avantage du *.Net Remoting* réside dans la flexibilité de son interface de communication. Il peut utiliser n'importe quel type de protocole (*HTTP* et *TCP* sont proposés par défaut) et d'encodage (des encodages binaires et XML/SOAP sont disponibles par défaut).

L'objectif du présent projet est de développer des outils afin d'analyser le comportement et la performance d'applications par objets répartis, en particulier les objets réseaux tels que définis dans le CLI (Common Language Infrastructure) de l'organisme de normalisation ECMA (European Computer Manifacturers Association) et utilisés avec le projet Mono. Ceci peut être effectué par une caractérisation précise de l'interaction entre tous les composants de l'application répartie et de l'utilisation des ressources de chaque composant.

Les composants interagissent par des messages que s'échangent des objets qui se trouvent soit sur le même système, soit sur des systèmes differents en réseau. L'originalité de l'outil développé se situe au niveau de son application au tout nouvel environnement .NET/Mono, de sa grande précision et de sa modeste pénalité en

performance.

Le deuxième chapitre du présente mémoire effectue une revue de la littérature des outils d'analyse de performance des objets réseau en étudiant les meilleurs systèmes et les solutions d'analyse existantes. Le chapitre trois présente les concepts et les composants du protocole *.Net Remoting*. Le quatrième chapitre décrit l'analyse proposée de performances du protocole *.Net Remoting* spécifique à l'environnement CLI. Le chapitre cinq est consacré aux expériences effectuées à l'aide des outils proposés. Suivent les conclusions.

CHAPITRE 2

REVUE DE LA LITTÉRATURE

2.1 Notions introductives

L'évolution des systèmes répartis et les performances des technologies Internet ont attiré les compagnies qui veulent partager les fonctionnalités de leurs applications (d'affaires ou de recherche scientifiques).
Le développement des systèmes répartis résulte du besoin de communication, de partage d'information et d'interopérabilité.
La plupart des projets ont comme aspects communs: le remodelage des anciennes applications pour transférer leur fonctionnalité vers l'environnement serveur, la conception de nouvelles interfaces cibles indépendantes pour les anciennes applications et la conception des nouvelles applications [41].
Des préoccupations liées à la sécurité, la fiabilité et la performance surviennent pour de telles applications.

Une *application répartie* [53] est composée d'un certain nombre de processus autonomes et séquentiels ou entités qui coopérent pour réaliser un but commun. La coopération inclut la communication et la synchronisation et est obtenue par l'échange de *messages*. Les échanges de messages synchrones ou asynchrones sont possibles. Les processus peuvent être créés et terminés dynamiquement.
La compréhension d'une application répartie est une tâche difficile. Souvent, pour bien comprendre l'exécution d'une application répartie, on utilise les diagrammes processus-temps.

La programmation orientée composant est établie comme une base pertinente pour le support de communications comprenant des composants matériels et logiciels hétérogènes. Des organismes internationaux comme ISO, ECMA et l'OMG

ont défini des normes pour les applications par objets servant de base pour le développement des systèmes répartis ouverts.
On trouve présentement trois modèles principaux de distribution d'objets:

- CORBA
- RMI
- .NET/Mono

Tous ces modèles se basent sur les mêmes principes, mais chacun d'eux présente des avantages particuliers.
Même si la comparaison entre différents modèles se fait en termes de caractéristiques, de maturité et de support pour les anciennes applications (*legacy system*), la facilité de developpement et les coûts cachés de maintenance sont des *performances* qui méritent une attention spéciale, surtout pour les applications scientifiques et d'ingénierie.
Les performances d'une application répartie sont souvent determinées par les performances de temps au niveau des couches de communication sous-jacentes. Le protocole *.NET Remoting* assure la communication entre différentes applications, que celles-ci résident sur le même ordinateur, ou sur des ordinateurs différents au sein du même réseau local ou sur des réseaux différents, chacune ayant son propre système d'exploitation.

Avant de détailler les concepts de base du *.Net Remoting*, une revue des différentes solutions utilisées dans les *anciens* modèles permettra de mieux apprécier les fonctionnalités existantes dans la nouvelle technologie .NET/Mono.
Les architectures mentionnées sont développées en utilisant les solutions actuelles plus matures: la *technologie orientée-objet* et le *patron client-serveur*. Une application répartie devient une collection d'objets qui interagissent. Un programme peut être à la fois client de certains serveurs et serveur d'autres clients.

Pour une meilleure compréhension d'une application répartie, dans le but d'analyser ses performances, on utilise la surveillance (*monitoring*) pendant l'exécution du processus.

Il y a trois aspects à surveiller pour tous les modèles d'objets répartis:

- la collecte des traces d'exécution;
- la gestion et le stockage des traces;
- l'analyse et la visualisation des traces.

La collecte des traces est réalisée par deux méthodes d'instrumentation:

- *L'échantillonnage*, qui consiste en une observation périodique de l'état du système et l'incrémentation des compteurs associés à l'état observé.
  Cette méthode n'est pas précise à 100% (des événements notables pour l'analyse peuvent être perdus entre les moments d'échantillonnage), mais la pénalité imposée au système est très basse. Elle est préférée pour les outils de profilage qui donnent une vue complète du système (OProfile [27]) ou du processus (gprof [10]).
- *Le traçage d'événements*, qui comprend la capture des séquences d'événements. Chaque événement correspond à une action physique/logique.

Habituellement la collecte de traces est réalisée par l'instrumentation du code source pour surveiller les événements importants pendant l'exécution.
L'instrumentation du code peut se situer à différents niveaux:

- niveau du système d'exploitation,
- niveau de l'environnement d'exécution,
- niveau de l'application.

Le temps de cette collecte d'information joue un rôle important dans les résultats finaux et son impact sur la synchronisation de l'application tracée doit être minimal. En effet, au niveau du temps d'exécution d'une requête, les intervalles se comptent en dizaines de millisecondes.
Les informations collectées peuvent être utilisés dans différents buts: le déboggage, la visualisation et l'analyse de performances. L'analyse post-mortem effectue le stockage des traces sur le disque dur, dans des fichiers ordinaires ou de format spécial.

La visualisation de données tracées diffère d'un système à l'autre, chacun ayant ses propres représentations: l'analyse de performance, l'analyse du trafic. L'exactitude des informations dans une trace dépend du niveau où est placé le code instrumenté qui les fournit. L'instrumentation du code au niveau application exige de connaître le code source de l'application et implique la modification complète de l'application, avec un impact majeur sur son comportement réel. L'instrumentation du code au niveau du système d'exploitation peut ne pas distinguer toutes les phases importantes du comportement de l'application répartie. Il reste une autre option viable: il s'agit d'introduire le code d'instrumentation au niveau de l'environnement d'exécution. Cette solution a été choisie dans ce projet.

Un problème déterminant dans les systèmes répartis est la gestion du temps. Il faut que toutes les horloges des composants du système réparti soient synchronisées afin de retracer les délais de chaque message échangé entre les objets. On ne peut pas synchroniser parfaitement les différentes horloges locales dans un réseau en utilisant NTP [22] et les horloges dérivent les unes par rapport aux autres.
Souvent, seulement l'ordre entre les événements importe; on parle ici de la synchronisation des horloges logiques [54].

Un système réparti est un ensemble de composants qui interagissent.
Un composant correspond à un objet ou à une agrégation d'objets au sens logiciel. Tous les modèles populaires (CORBA, Java, .NET/Mono) sont des architectures

orientées objet. Le changement d'état d'un objet représente une action. Les actions associées à un objet sont décomposées en: actions internes et interactions [62].
Une *action interne* représente un changement d'état spontané de l'objet, sans intervention de l'extérieur tandis qu'une *interaction* réfère à un changement d'état provoqué par l'extérieur de l'objet via une de ses interfaces. L'objet est encapsulé, cela signifie que son état n'est pas observable de l'extérieur à moins qu'il ne possède une interface dédiée à cet usage, ou que le support d'exécution offre la réflexivité (Java, CORBA, .NET). Chaque interaction est reliée à une interface unique.
Une interaction entre interfaces est possible seulement si un lien a été établi. Un lien représente un chemin de communication entre interfaces ou canal. Le canal est soit un tampon FIFO parfait, ou un lien qui peut perdre, réordonnancer et/ou dupliquer les messages.
Les ressources utilisées par tous les processus d'une application répartie sont classées en deux catégories: les ressources de calcul et les ressources d'interaction.
Nous appelons *ressource* [62] tout objet pouvant être utilisé par un processus. À une ressource sont toujours associées des procédures d'accès, qui permettent de l'utiliser, et des règles d'utilisation qui constituent son *mode d'emploi.*
Les ressources correspondent aux entités de base sur lesquelles repose l'exécution. Le partage d'une ressource peut générer des situations de privation ou d'interblocage et est souvent à l'origine des problèmes de performances.
Les *ressources de calcul* correspondent à l'ensemble des moyens mis en œuvre pour que les actions de traitement s'exécutent bien. Les *ressources d'interaction* correspondent aux données de l'objet qui sont à l'extérieur. L'accès à une donnée de l'extérieur de la capsule de l'objet n'est possible que par une interaction qui dépend elle-même de la création d'une liaison.

Dans toutes les architectures (CORBA, RMI, .NET) on trouve la même organisation des applications réparties utilisant l'intergiciel.

L'*intergiciel* (fig. 2.1) est une couche de logiciel (répartie) destinée à:

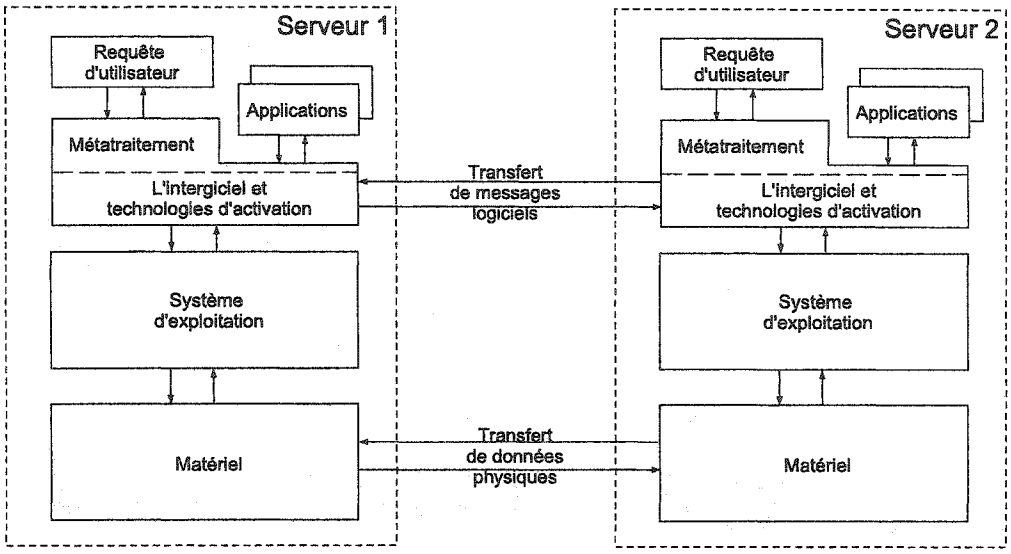

Figure 2.1 L'intergiciel et le contexte environnant

- masquer l'hétérogénéité des machines et des systèmes,
- masquer la répartition du traitement et des données,
- fournir une interface commode aux applications (modèle de programmation).

La construction des applications réparties est facilitée par l'utilisation d'une architecture déjà développée (CORBA, Java RMI, .NET/Mono) à partir de composants simples. Tous les cadres utilisent les appels de procédures à distance, inspirés des Sun RPC [58], pour exécuter:

- l'activation à distance des objets;
- l'invocation à distance des objets.

Le RPC maintient une certaine transparence; du point de vue du client, la procédure appelée apparaît comme une procédure locale. En fait, la procédure distante appelée est représentée localement par une procédure particulière appelée *souche*

cliente; c'est cette procédure locale que le client invoque réellement. Cette souche cliente utilise ensuite le réseau pour contacter la procédure distante implantée par le serveur.
Un module particulier contrôle l'accès aux procédures du serveur, la *souche serveur*. La communication entre la souche cliente et la souche serveur se fait indirectement via deux services:

- le service XDR (eXternal Data Representation) qui assure la mise en forme des paramètres échangés (codage, élimination des pointeurs);
- le service RPC qui envoie les requêtes, reçoit les réponses et gère les erreurs.

Les échanges à travers le réseau peuvent se faire via TCP ou via UDP. Les Sun RPC ont été conçus pour la programmation procédurale et ne prévoient rien en ce qui concerne la programmation par objets. L'adaptation des RPC à la programmation par objets s'appelle RMI (Remote Method Invocation).
S'il existe des RMI qui peuvent s'adapter aux objets Java, le langage Java fournit un RMI qui est propre aux objets Java. Il s'agit ici de travailler dans un environnement homogène Java; contrairement à CORBA qui prévoit un environnement hétérogène, permettant des langages différents.

Au delà des principes conceptuels présentés ci-dessus, on a souvent besoin de connaître les aspects détaillés du fonctionnement interne d'une certaine architecture, pour améliorer et ainsi pour contrôler (qui implique la mesure exacte) les performances d'une application répartie. De ce point de vue on peut faire une distinction entre les solutions commerciales (pour lesquelles les mécanismes internes ne sont pas toujours disponible) et les solutions à code source libre.
Une revue des systèmes existants (RMI, CORBA, Remoting) est présentée dans les sections suivantes en décrivant pour chaque système les mécanismes d'obtention des mesures/données, ainsi que les algorithmes et les techniques de présentation des résultats utilisés. Des recherches antérieures [34] ont mesuré les performances

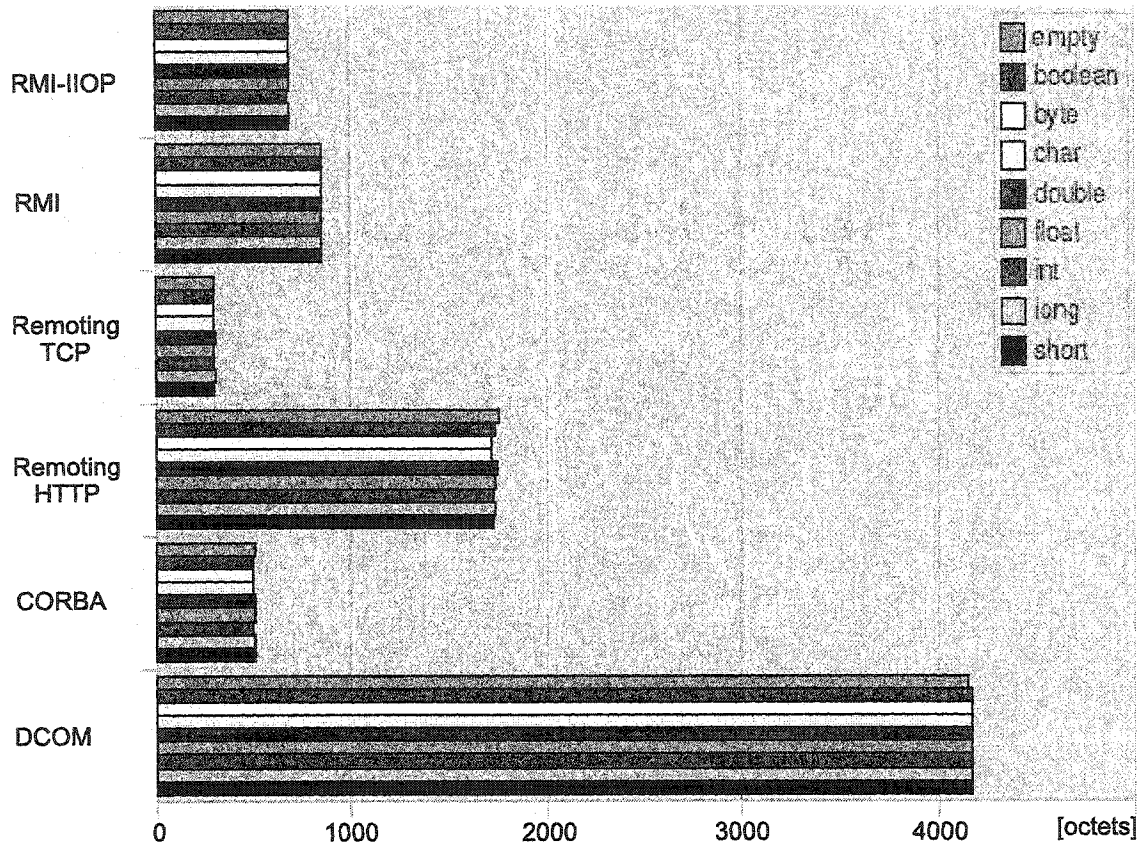

Figure 2.2 La pénalité d'un accès réseau pour des arguments de types primitifs dans le cas de différents protocoles sous-jacents [34]

des intergiciels (fig. 2.2) étudiés en fonction de certains paramètres.

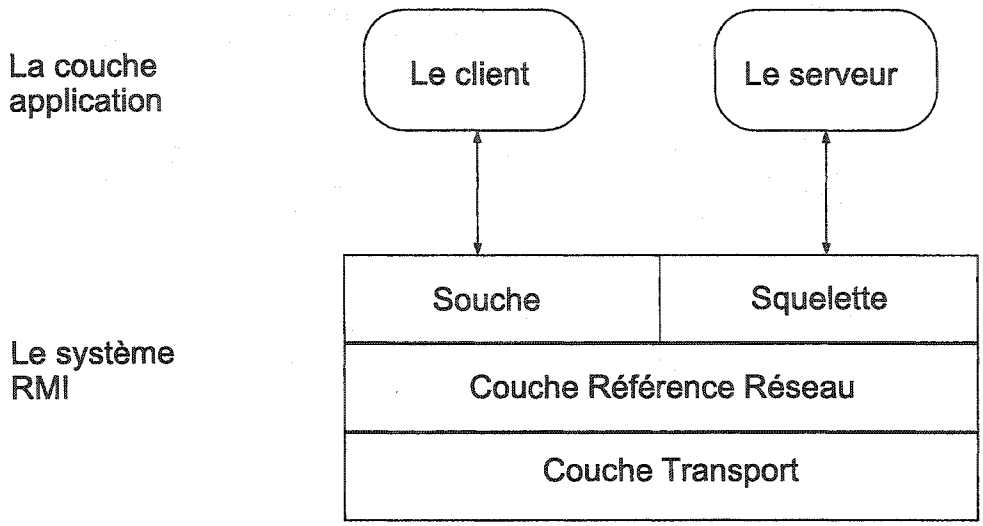

Figure 2.3 L'architecture RMI

2.2 Java RMI

JavaRMI (fig. 2.3) est l'API de manipulation d'objets répartis en Java. À partir de l'interface d'un objet distant à accéder, sont générés: une *souche* et un *squelette*. Les deux entités sont générées par un compilateur RMI (RMI *compiler* ou *rmic*) fourni dans l'environnement Java.

L'objet serveur s'enregistre dans un annuaire afin d'être localisable par ses clients (*RMI registry*). Les clients trouvent l'objet distant et appellent ses méthodes via la souche installée de leur côté. Tout objet Java publiant son interface pour des clients distants implémente *java.rmi.Remote*.

La seule différence dans les appels d'objets est que toute méthode est susceptible de lever une exception: *java.rmi.RemoteException.*

```
public interface ServeurAddition extends java.rmi.Remote {
        public int somme(int a,int b) throws RemoteException;
}
```

L'architecture *RMI* comprend trois couches sous-jacentes indépendantes: la couche souche/squelette, la couche référence réseau et la couche de transport. Au niveau de la couche de transport, on retrouve deux alternatives:

- JRMP, conçu pour la communication entre objets Java uniquement et optimisé pour eux (basé sur les protocoles: Java Object Serialization et HTTP ou directement les interfaces de connexions, *sockets*, offerts par le système d'explotation).

- IIOP (tel que défini pour CORBA, on parle alors de RMI/IIOP) pour garantir l'interopérabilité avec d'autres systèmes, communiquer avec des objets non-Java, transporter des informations techniques comme un contexte de sécurité ou un contexte transactionnel.

La communication client/serveur est implémentée par la classe souche du côté client et par la classe squelette du côté serveur.
La couche souche/squelette est l'interface entre la couche application et le système RMI. Cette couche transmet les données vers la couche de référence du contrôle à distance par le biais d'une abstraction en *flot de conversion* (*marshal stream*).
La conversion est la transformation d'une donnée vers un format standard qui permet de véhiculer cette représentation à travers des réseaux/hôtes sans altérer la valeur. Par exemple, dans la version Sun Java development kit 1.3, pour que RMI soit implémenté au dessus de IIOP, la conversion consiste à transformer le résultat de sérialisation dans une valeur CORBA CDR [8]. La sérialisation est un algorithme ou une technique qui permet de transformer une structure de données quelconque sous une forme sérielle/séquentielle. Depuis la version Sun Java 1.2, les squelettes ne sont plus nécessaires, la réflexivité est utilisée (*java.reflect*) [57].

La plupart des outils d'analyse de performance pour les applications réparties développées en Java utilisent la génération de trace comme méthodologie de base. *Log4j* [18], fait partie du projet Apache et fournit une bibliothèque flexible pour

journaliser les applications. Cependant ses performances sont loin de répondre aux exigences d'une surveillance détaillée.

Un autre paquet d'instrumentation a été développé par Hewlett-Packard et Tivoli: *ARM* (Application Response Measurement) API [60] qui fournit la surveillance par l'encapsulation des appels d'un logiciel qui peuvent être capturés par un agent qui supporte ARM API. Cet API sert à mesurer le temps UCT passé dans chaque section de l'application. L'utilisation de cet API a un impact sur l'architecture de l'application.

2.2.1 ProActive - IC2D

Interactive Control & Debug for Distribution

IC2D [14] du laboratoire français I.N.R.I.A. [13] permet de surveiller, de contrôler et de visualiser graphiquement les communications d'une application répartie, indépendamment de son environnement d'exécution, de la station de travail multiprocesseurs à une grappe (en utilisant les outils Globus [9]). IC2D s'avère un outil efficace pour la métaprogrammation: lancement des tâches, affichage et surveillance de la topologie des communications, migrations des tâches par glisser-déposer.
L'équipe OASIS [6] développe des solutions dans le cadre des applications réparties: techniques et outils pour la construction, l'analyse, la validation et la maintenance des systèmes fiables. Dans le cadre du projet GRID RMI, la tâche consiste à expérimenter l'utilisation de l'interface graphique IC2D comme pilote d'application en utilisant Padico (environnement développé à l'IRISA [12]), afin d'exploiter efficacement les ressources réseaux.
Toute application répartie doit s'exécuter dans un environnement lui donnant accès à un certain nombre d'entités ou ressources tant matérielles que logicielles. De telles entités sont utilisées par le biais d'interfaces prenant la forme d'un service. Afin de pouvoir utiliser une entité, il est nécessaire de la nommer. Si on n'en connaît pas le nom, il est nécessaire de parvenir à le découvrir. Jini est une bibliothèque proposée par SUN pour la programmation distribuée, où la découverte des entités distantes se fait *par type* à l'opposé des systèmes classiques où elle se fait *par nom*. Jini souffre cependant d'une certaine rigidité.

IC2D est construit par dessus RMI et ProActive, avec lesquels il est possible de faire des appels asynchrones. *ProActive* est une librairie 100% Java, développée pour la programmation parallèle, distribuée et concurrente. ProActive fournit un service d'invocation de méthodes à distance vers des objets actifs distribués de façon transparente, de communications asynchrones et de mécanismes de synchronisation haut-niveau.

ProActive est bâtie sur les APIs standards de Java et ne nécessite aucune modification de l'environnement d'exécution.

Une application répartie et/ou concurrente construite à l'aide de ProActive est composée d'entités de granularité moyenne appelées *objets actifs*. Chaque objet actif a son propre fil d'exécution qui possède la capacité de décider dans quel ordre il doit servir les appels de méthodes, qu'il reçoit et stocke dans une file d'attente de requêtes. ProActive a la capacité de créer des objets actifs distants. Pour cela, il faut être capable d'identifier la JVM et fournir quelques nouveaux services. Les nœuds sont des objets définis dans ProActive dont le but est de recueillir plusieurs objets actifs dans une entité logique. Ils fournissent une abstraction pour la localisation physique d'un ensemble d'objets actifs. La manière traditionnelle d'appeler et de manipuler les nœuds est de leur associer un nom symbolique. Celui–ci est l'URL spécifiant leur localisation (rmi://lo.inria.fr/Node1).

Avec l'interface graphique (fig. 2.4) il est possible de suivre tous les objets actifs, les états des objets (en exécution ou en attente d'une requête ou de données), les événements et les liaisons entre eux (envoi-réception). Dans le module de contrôle, il est possible d'exécuter l'application au fur et à mesure.

IC2D fournit beaucoup de caractéristiques graphiques et interactives. IC2D est présentement utilisé dans les applications employant EJB, ordinateurs en grappe et aussi comme base de gestion de système et de réseau.

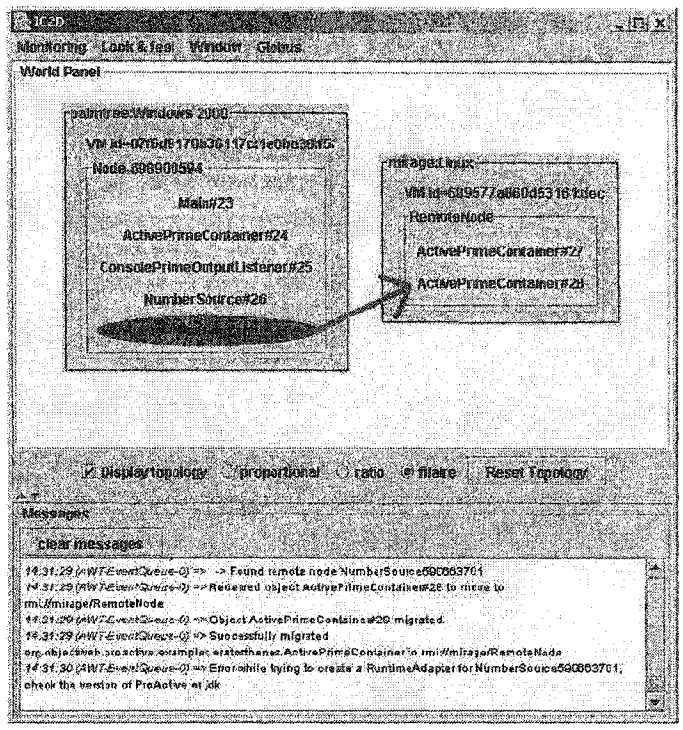

Figure 2.4 IC2D: vue du journal pour un échange de messages

2.2.2 JaViz

L'outil de visualisation d'exécution de JaViz [20] est conçu pour l'analyse des performances des applications réparties et pour l'identification des méthodes inefficaces.

Les applications client/serveur à grande échelle distribuent des objets et ensuite s'exécutent à travers des machines multiples. Une étape critique dans l'exécution de ces applications réparties est l'identification des *points chauds* où il y a une invocation à distance fréquente de la méthode (RMI).

Les méthodes codées pour la flexibilité et la généralité peuvent poser des problèmes significatifs d'exécutions une fois utilisées intensivement dans des applications plus larges. Par exemple, les réalisateurs découvrent souvent qu'une quantité excessive de temps est passée dans certaines méthodes d'espace de nom *Java.String*.

JaViz a été développé pour compléter, plutôt que remplacer, les outils d'analyse

existants en Java, tels que l'outil HyperProf [19], JProbe [30], et OptimizeIt [7].
Il n'est pas possible de tracer des fils d'exécution à travers des frontières de JVM, quand s'exécutent des appels de RMI. Les outils JProbe et OptimizeIt fournissent des analyseurs graphiques puissants pour identifier les goulots d'étranglement à l'exécution dans les programmes Java, mais ils ne soutiennent pas le traçage coordonné des activités client/serveur.
Jinsight [11] est un autre outil de visualisation d'exécution pour les programmes d'application Java. Jinsight emploie des traces d'une machine virtuelle Java modifiée, pour montrer les goulots d'étranglement à l'exécution, la création d'objets, le ramasse miette, l'interaction des fils, et les références d'objet.
L'application Jinsight d'IBM, n'offre pas la possibilité de tracer des activités client/serveur à travers plusieurs JVM.
Jinsight est très utile pour l'analyse de l'exécution des applications Java fonctionnant sur un simple JVM, mais il ne peut pas être employé pour identifier les goulots d'étranglement à l'exécution dans les programmes d'applications distribués en Java.

JaViz est constituée de trois composants principaux: un *JVM* modifié pour tracer les appels de méthode, un *ensemble d'outils* de post-traitement qui résoud les appels de la méthode à distance et qui produit des statistiques et un *outil de visualisation.*

Le visualiseur lit les données de l'étape de post-traitement, qui contiennent l'arbre dynamique d'exécution d'un programme, et les montre graphiquement comme arbre avec l'information détaillée dans chaque nœud.
Le module de génération de trace du JVM est modifié pour enregistrer chaque invocation d'une méthode, en utilisant les temps de début et de fin de la méthode avec une résolution d'une micro-seconde (fig. 2.5). De plus, une étiquette *thread_ID* est enregistrée pour identifier uniquement le fil d'exécution de la méthode.
Parfois le nombre de méthodes appelées dans un programme peut être grand, et chaque instance d'une méthode produit une entrée de trace. La quantité de données

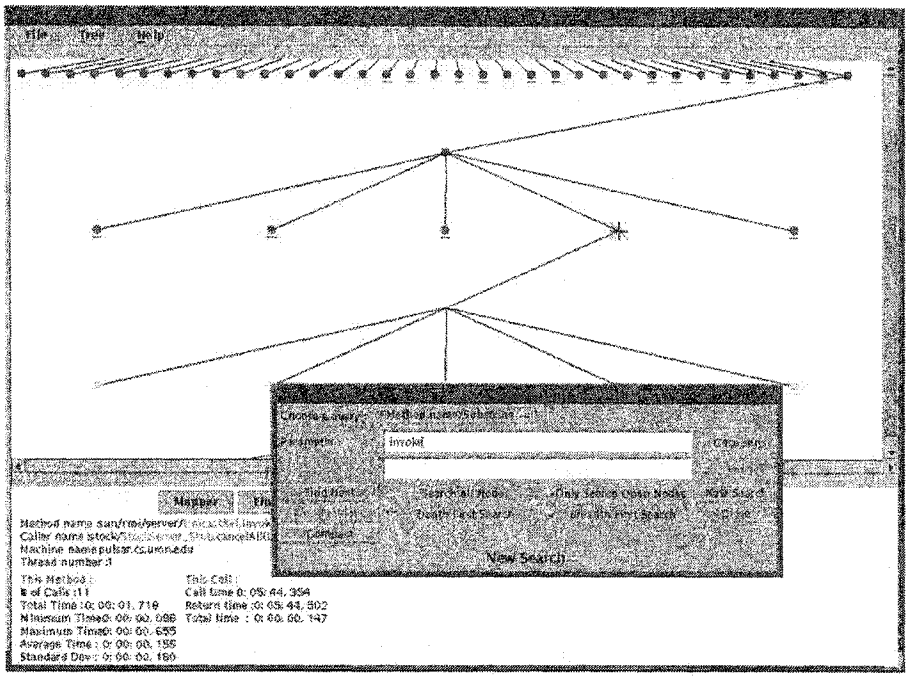

Figure 2.5 JaViz: détailles journalisés pour l'appel d'une méthode

de trace produites pour une grande application est énorme. Des options de filtrage sont prévues pour une JVM instrumentée, dans le but de réduire la quantité de données tracées.

L'outil d'analyse d'exécution JaViz a été développé pour résoudre les problèmes d'exécution d'applications Java distribués à grande échelle. Les traces pour différents fils d'exécution fournissent des moyens pour analyser les applications multi-fils, alors que les traces des appels RMI permettent d'identifier les *points chauds* dans les applications client/serveur.

2.3 CORBA

CORBA est la spécification d'une architecture orientée objet pour applications réparties. Elle a été définie par l'Object Management Group (OMG) [23] dans un document publié en 1990 avec pour objectif de: *faire émerger des standards pour l'intégration d'applications réparties hétérogènes et la réutilisation des composants logiciels.*

Pendant 13 années, l'organisation OMG a développé trois versions finales des spécifications de plus en plus performantes. CORBA est un standard ISO (ISO/IEC 19500-2): CORBA 1.0(novembre 1991), CORBA 2.0(août 1996), CORBA 3.0(septembre 1999).

CORBA est composé d'un *Object Request Broker* (ORB) utilisé pour le transport des requêtes et l'activation des objets, des *services de base* horizontaux (CORBA_services), des *utilitaires communs* verticaux (CORBA_facilities) et des *interfaces* de domaines(*business object*).

L'ORB est un intergiciel, qui établit la relation client–serveur entre les objets et standardise la communication entre les objets distants:

- indépendamment du langage
- indépendamment de la plate-forme.

L'un des nombreux avantages de CORBA est qu'il supporte plusieurs langages (fig. 2.6). En utilisant un ORB, un client peut invoquer, d'une façon transparente, une méthode sur un objet serveur, qui peut être sur la même machine ou sur une machine distante.

Le client et le serveur sont deux objets CORBA qui communiquent au moyen d'invocation de la méthode. Le client appelle une méthode sur le serveur à distance par simple invocation *objet.methode(args)*. Le serveur renvoie le résultat comme une valeur de retour ou par des arguments. L'ORB cache la localisation de l'objet,

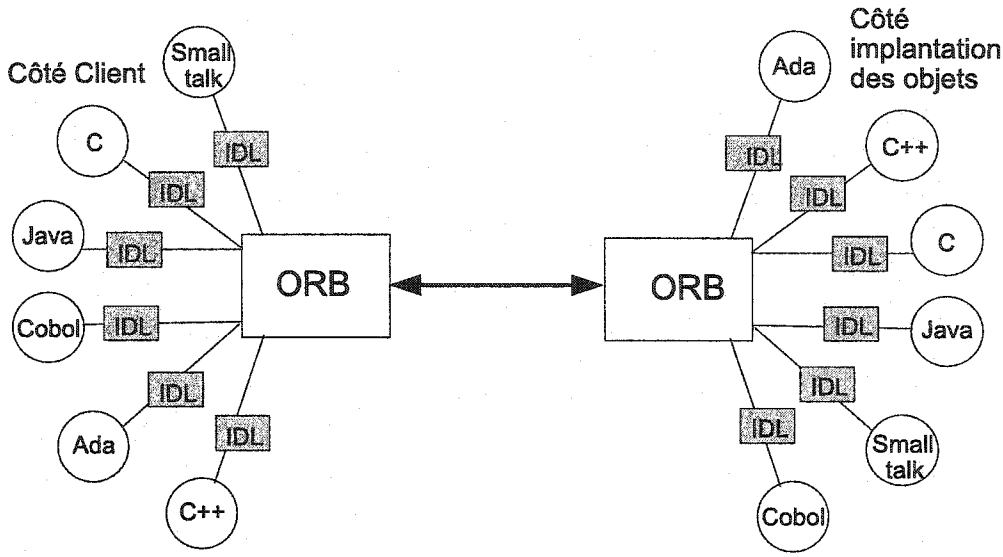

Figure 2.6 CORBA esperanto

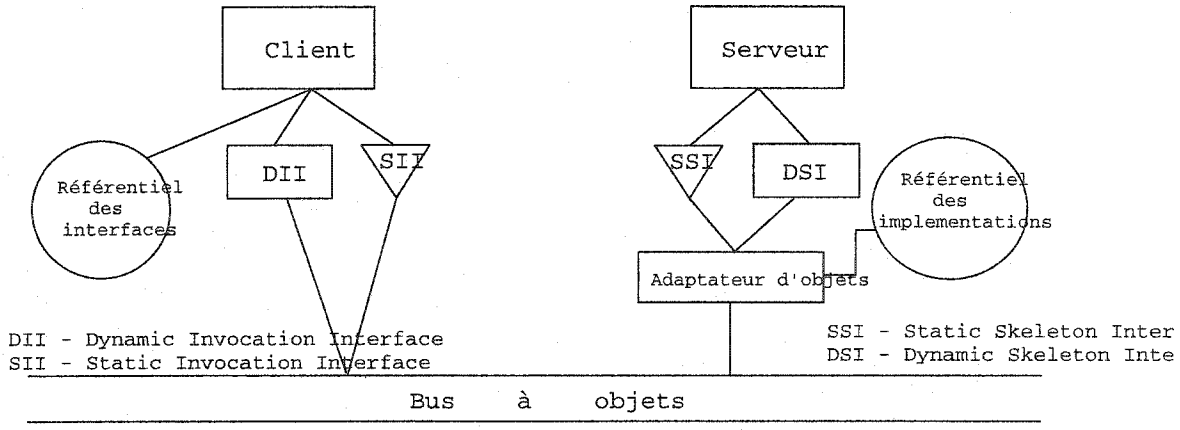

Figure 2.7 CORBA: les composants

l'implémentation, l'état d'exécution et les mécanismes de communication. L'ORB soutient deux types d'invocation de méthode (fig. 2.7):

- l'invocation statique - le client doit connaître les informations relatives à l'interface des objets au moment de la compilation
- l'invocation dynamique - permet la communication entre les objets sans connaître les méthodes au moment de la compilation. Les informations sur les interfaces sont stockées dans le dépôt d'interfaces (*Interface Repository*).

L'interface d'un objet serveur doit être écrite en IDL, un outil de spécification des interfaces, indépendant du langage de programmation. La compilation de cette description IDL génère la souche, une librairie utilisée par le client pour invoquer les opérations du serveur, et le squelette, une librairie associée au serveur. Il y a un grand nombre d'implémentations CORBA. Chaque implémentation comprend généralement:

- un compilateur IDL qui permet de convertir le langage IDL vers un ou plusieurs langages cibles,
- un *ORB* qui traite les requêtes CORBA; un bus commun pour tous les objets CORBA;
- quelques Services: le service de nom, le service d'accès concurrent, le service de transactions, le service de temps, le service de sécurité et le service de cycle de vie.

La communication à distance utilise le protocole GIOP (General Inter-ORB), qui constitue un protocole de haut niveau, et la couche de transport sous-jacente indépendante. Les standards CORBA spécifiant le GIOP définissent le format des données (CDR), le format de message GIOP et les paramètres de transport GIOP. IIOP(Internet Inter–ORB Protocol) est l'implémentation du GIOP sur TCP/IP et spécifie comment les agents ouvrent les connections TCP.

Exemples d'implémentations CORBA:

- commerciales: Iona Orbix [17], VisiBroker [32], ORBAcus [28] (Open Source)
- libres: omniORB [24] (licence GPL/LGPL), ORBit [29] (licence GPL/L-GPL), TAO [5].

CORBA permet l'intégration d'applications déjà existantes avec des nouvelles technologies; une application COBOL peut être interfacée par un objet CORBA et utilisée par des clients Java.
CORBA offre une grande souplesse pour faire évoluer une application, en taille et en fonctionnalités, et aussi une robustesse à l'utilisation. L'ORB fournit la flexibilité, tout en permettant au programmeur de choisir le système d'exploitation, l'environnement d'exécution, et même le langage de programmation à utiliser pour chaque composant d'un système en construction.

2.3.1 Le traçage d'une application répartie sur CORBA

Les trois aspects à surveiller (*la collecte*, *la gestion* et *la visualisation des traces*) se retrouvent dans les applications réparties construites en utilisant CORBA.
David A. Carr propose une étude [35] dont le but final est la création d'un outil pour l'analyse et la mesure des performances d'une application répartie. Le traçage de RPC est fait avec les intercepteurs CORBA. Les données tracées sont analysées pour re-construire les séquences requête – réponse et un outil commercial (Spotfire) est utilisé pour visualiser les statistiques et le comportement anormal.

- *Étape 1* Le traçage et l'impact minimum sur l'application. Les intercepteurs CORBA fournissent des crochets (*hooks*) au niveau RPC. Un mécanisme d'interception est ajouté, sur le client CORBA ou sur le serveur CORBA, pendant la phase d'édition de liens dans le cas du C++, ou à l'exécution pour les applications en Java.
  Un aspect très utile est la possibilité d'ajouter/enlever dynamiquement les points d'interception. L'implémentation du mécanisme de traçage diffère pour les deux solutions CORBA choisies (Orbix de Iona Inc et Visibroker de Inprise Inc). Les différentes implémentations ORB ne sont pas interopérables à 100%.

- *Étape 2* Générer les séquences d'interactions et construire la trace des événements du point de vue logiciel. Finalement, des statistiques pour différentes périodes de temps (par heure, par jour, par semaine, par mois) sont fournies. Pour chaque période, les paramètres suivants sont calculés:

  - le temps de réponse;
  - le nombre d'appels d'une méthode (méthode terminée ou non, avec l'exception);
  - le nombre d'appels entre un client et un serveur;

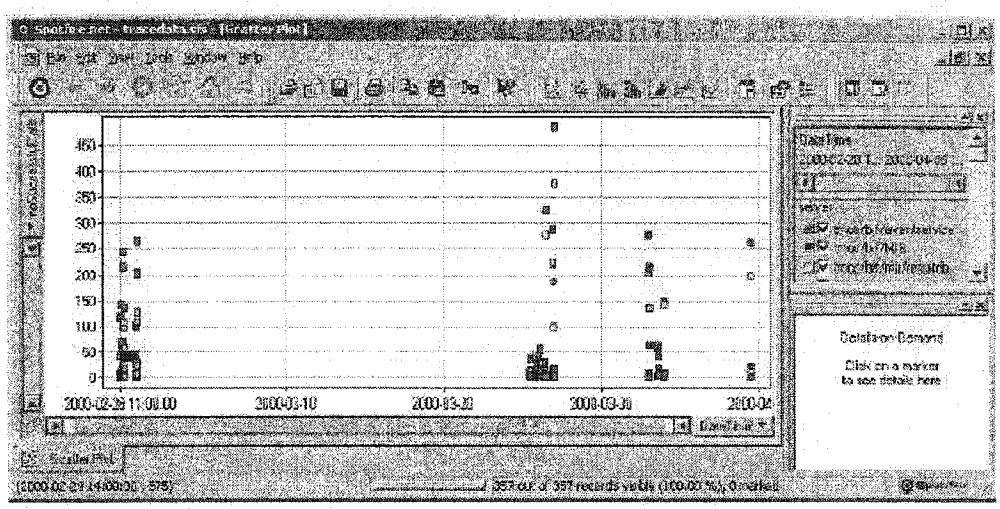

Figure 2.8 L'analyse de traçage d'interactions dans l'application CORBA

  - la quantité de données envoyées.

- *Étape 3* L'analyse statistique du traçage peut se faire en utilisant n'importe quel outil graphique. L'exemple proposé par les auteurs est Spotfire.

2.4 NetLogger Toolkit

NetLogger a été développé dans les Laboratoires Lawrence Berkeley National (University of California) [38] pour surveiller le comportement de tous les éléments qui composent un système complexe (de bout en bout) et les voies de communication entre les applications, afin de déterminer avec précision où le temps est passé dans un système réparti.
Il a été développé avec comme objectifs de fournir:

- des outils pour faciliter les applications réparties qui journalisent des événements intéressants à chaque point critique;
- des outils pour la surveillance du réseau.

La méthode combine le réseau, le client et la surveillance au niveau de l'application pour fournir une vue complète du système. On observe souvent de faibles performances dans les applications réparties. Les goulots peuvent être dans les composants suivants:

- les applications;
- les systèmes d'exploitation;
- les disques ou les adaptateurs de réseau (pour les opérations: *requête/réponse*);
- les commutateurs ou les passerelles du réseau.

Le paquet NetLogger propose des outils d'analyse qui fournissent des informations en temps réel. Parmi les causes des problèmes de performances se trouvent:

- 40% de problèmes de réseau;
- 20% de problèmes de processeur;

- 40% des problèmes d'architecture d'application (50% côté -serveur+50% côté -client).

NetLogger ToolKit contient les composants suivants:

- une API et une bibliothèque pour la génération des traces au niveau utilisateur ;
- les outils pour la collection et la manipulation des traces;
- les outils de surveillance pour client/serveur;
- les outils de visualisation (*NLV*).

Un composant additionnel est NTP [22] (Network Time Protocol) pour la synchronisation du temps sur tous les systèmes.

La capacité de NetLogger de corréler les données surveillées par l'instrumentation de l'application avec le trafic de données à travers le réseau, pour déterminer les *responsables* des goulots, est une technique utile d'analyse et de déboggage.

Afin d'instrumenter une application pour obtenir des traces, des appels vers NetLogger API sont ajoutés dans les points critiques du code, l'application étant liée à la bibliothèque NetLogger.

```
log = NetLogger(program_name,x-netlog://loghost.lbl.gov);/* instance */
...
log.write(EVENT_START,TEST.SIZE=%d,size); /*insertion de trace*/
```

Aujourd'hui, les bibliothèques sont disponibles pour les langages: Java, C, C++, Python, et Perl. Les messages générés sont en format binaire ou ASCII, tel que prescrit dans les spécifications ULM (Universal Logger Message). Dans les traces d'applications réparties (fig. 2.9) on peut ajouter des données collectées avec des outils standards Unix (vmstat, iostat, netstat, snmpget) pour avoir une image globale de la charge de chaque pièce d'une application répartie. Les composants de NetLogger Toolkit sont sous licence Open Source.

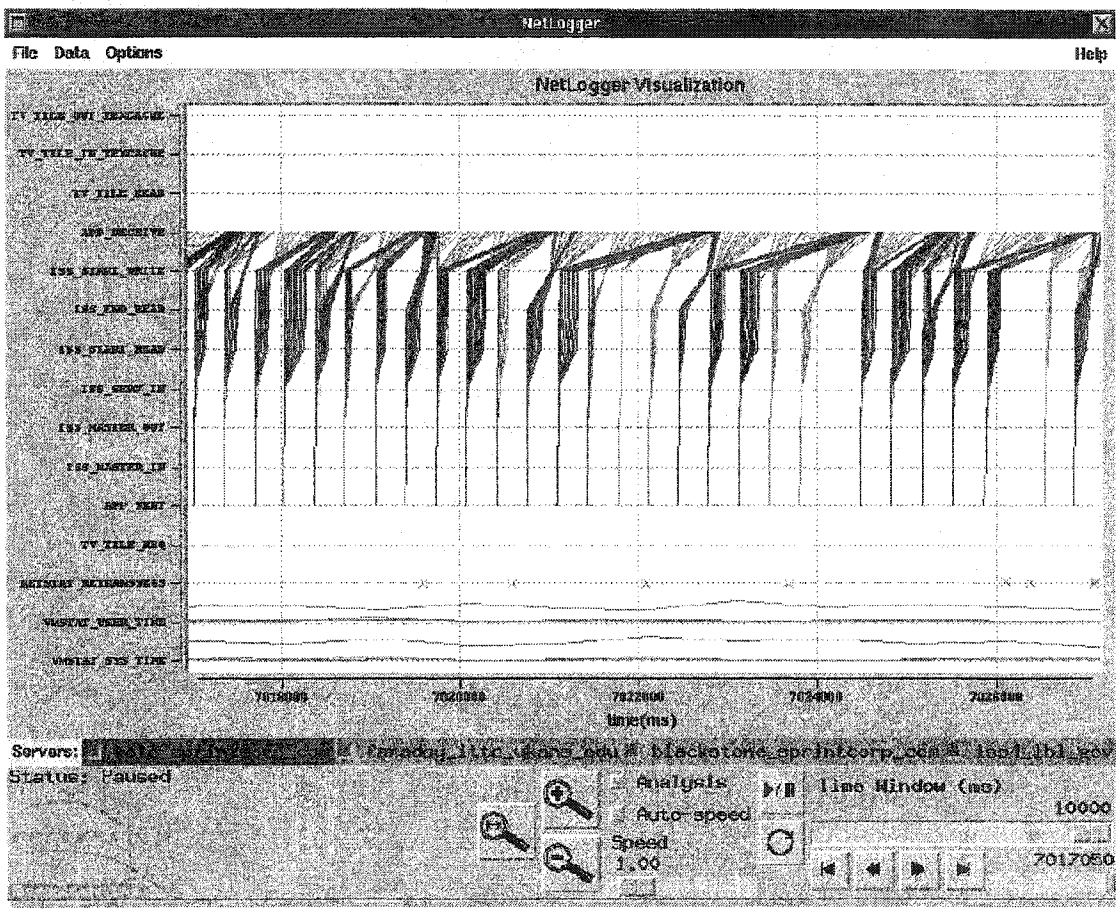

Figure 2.9 NetLogger: visualisation d'interactions entre processus

2.5 Les outils commerciaux et les travaux apparentés

Un certain nombre de systèmes offrent des fonctionalités intéressantes mais sans que leur fonctionnement ne soit documenté. Les plus intéressants sont discutés ici en dépit du peu d'information disponible.

Borland Optimizeit Suite 5.5 pour Java

La solution JBuilder est un environnement qui comprend toutes les phases du cycle de développement pour une application.

- Optimizeit Profiler [7] est un des profileurs de code les plus utilisés dans la communauté des développeurs Java.

- Optimizeit Suite comprend des outils de mesure des performances, qui permettent aux développeurs d'améliorer la qualité de toute application Java client et/ou serveur, en terme de vitesse, robustesse et délai de livraison.

- Optimizeit Profiler permet de mesurer et de résoudre les problèmes de performance liés aux fuites de mémoire et aux goulots d'étranglement.

- Optimizeit Code Coverage permet d'assurer la fiabilité des tests et d'identifier les parties de codes non utilisées.
  Optimizeit Suite est disponible sur plates-formes Windows depuis octobre 2001.

Optimizeit Suite, Optimizeit Code Coverage et Optimizeit Thread Debugger seront disponibles sur Linux, Sparc Solaris et Solaris Intel.

Sitraka Software

Sitraka Software [30] vient de relâcher JProbe Suite 5.0, une suite d'outils de test de code Java. JProbe Suite offre des capacités d'optimisation des servlets, EJB et applications serveur. Ce système regroupe de puissantes fonctions de mesure des

performances, de débogage de la mémoire, de test du code et d'analyse des processus au sein d'une suite intégrée pratique, conçue pour le développement Java côté serveur.

JProbe Suite présente sous une forme graphique intuitive tous les aspects allant de l'utilisation de la mémoire jusqu'aux liens de dépendance, ce qui permet de remonter avec facilité et rapidité jusqu'à la source du problème.

Des versions pour Windows et Solaris sont disponibles, et s'intègrent facilement avec IBM VisualAge for Java.

Rational Rose Quantify

Quantify [31] est un produit commercial mature qui bénéficie d'un certain succès, surtout par sa disponibilité pour les deux systèmes d'exploitation Windows et Unix. L'outil permet de faire des tests de profilage pour évaluer les performances, avec support pour les applications .NET.

Les caractéristiques de l'outil:

- identifie les goulots d'étranglement dans l'exécution d'application,
- permet le traçage de tous les composants d'une application, avec ou sans le code source,
- l'analyse rapide pour tous les exécutables (applications VB, C, C++, C# et Java).

Travaux apparentés

ObjectWeb [25] est un projet initié et soutenu par France Telecom R&D, Bull et INRIA , ayant comme but le développement d'intergiciel réparti comme composants flexibles et adaptables. La disponibilité du code source de l'application est requise pour améliorer sa fiabilité et ses performances.

Le projet de recherche POET (Partial Order Event Trace) [40] a commencé comme débogueur pour les applications réparties. La définition actuelle du projet

est l'instrumentation des applications réparties pour collecter les événements avec lesquels on peut faire une trace d'exécution.

2.6 Les limites des approches proposées

Le développement des applications réparties fiables est une tâche difficile pour les architectes de systèmes. Obtenir des performances supérieures à celles des systèmes existants exige une compréhension détaillée de tous les événements échangés entre les composants pendant l'exécution.
Les outils d'analyse de performance jouent un rôle important dans cette compréhension. Parmi plusieurs facteurs qui influencent les performances des systèmes répartis se trouvent: les composants logiciels utilisés, la vitesse et le trafic du réseau, et le matériel de l'ordinateur local (UCT, sous-système d'entrée-sortie, mémoire).

> *La largeur de bande est une question de coût. La latence est plus difficile à améliorer parce que la vitesse de la lumière est constante – on ne peut pas acheter Dieu.* (David Clark – M.I.T.)

Les outils d'analyse sont vraiment essentiels pour la mesure et l'optimisation du temps de latence dans les interactions entre les composants d'un système réparti.

> *Deux facteurs significatifs affectent les processus en interaction dans un système réparti: la* performance de communication *est souvent une caractéristique limitative et il est impossible de définir une* notion de temps global unique [36].

L'objet de cette étude est d'analyser les performances de la communication entre les entités d'une application répartie, qui dépendent essentiellement du protocole sous-jacent utilisé. Tous les outils présentés ci-dessus ont comme but l'analyse des performances des applications réparties et la détection des raisons qui mènent à la dégradation de ces performances.

Les meilleurs sont ceux développés avec les architectures CORBA et Java, en utilisant les protocoles IIOP et Java RMI. Le protocole utilisé affecte les performances de communication. L'analyse évolutive et comparative entre ces deux

protocoles a aussi fait l'objet d'autres études [51, 1].
Les avantages et les inconvénients de chaque structure se réflètent dans les outils de performance présentés ci-dessus. Java RMI est un protocole spécifique à Java et s'utilise exclusivement dans cet environnement. RMI est moins efficace que la communication par les interfaces de connexions (*sockets*) à cause de la couche intermédiaire et de la pénalité du système pour accéder l'annuaire. RMI est dépendent de ses capacités de sérialisation pour transformer l'objet vers une représentation appropriée à la transmission. Il y a différentes solutions pour accélérer cette procédure [55]. RMI est dépendent d'un outil de compilation additionnel (*rmic*).
L'analyse de la pénalité des systèmes identifie des goulots d'étranglements comme: la mauvaise gestion des fils d'exécution, les algorithmes inefficaces pour le démultiplexage, des copies excessives de données, trop d'invocations de méthodes locales et architecture sans cache pour l'information fréquement demandée [51].

En comparaison avec d'autres outils d'analyse Java, *JaViz* [20] introduit une pénalité d'exécution de 1.5 fois plus grande à cause du système de traçage, ce qui implique un impact sérieux sur le comportement réel du système.
Les fichiers de trace sont volumineux (17 MB pour 30 minutes d'exécution) mais le support binaire est prévu.

Pro-Active IC2D [14] est un outil d'analyse pour applications Java seulement, ce qui constitue aujourd'hui une limitation dans un environnement fortement hétérogène. Les temps d'observation et d'affichage pour les evenements passés ont une précision de l'ordre des secondes et ne permettent pas l'analyse détaillée pour les systèmes dont les messages s'échangent avec une haute fréquence. En interne, l'outil emploie le paquetage Log4J, qui n'est pas un système de traçage rapide (introduit des délais dans l'exécution du processus analysé).

Sun a choisi IIOP comme protocole standard pour l'architecture EJB , ce qui implique que les applications Java de large dimensions possèdent un support et une maturité semblable aux spécifications CORBA. Cependant, Java supporte seule-

ment une partie de toutes les applications réparties existantes.

Les autres applications adoptent la solution offerte par OMG, CORBA, plutôt que celles des anciens systèmes (*legacy systems*) et des applications en temps réel. La complexité de CORBA affecte les performances des applications.

Les objets CORBA ne sont pas gérés par un algorithme ramasse–miette distribué et ils ne quittent jamais leur hôte d'implémentation. Ceci peut être considéré comme un désavantage car, une fois créés, les objets survivent jusqu'à la décision de les détruire (tâche difficile).

Une analyse comparative montre la bonne performance de CORBA dans le cas d'une application qui supporte plusieurs clients (pour huit clients RMI est 70% plus lent que CORBA [51]).

Pour l'outil d'analyse CORBA présenté plus haut [35], la division en trois composants complètement indépendants facilite la possibilité de remplacer chacun d'eux (par exemple le remplacement de l'outil de visualisation) sans en affecter la fonctionnalité.

Néanmoins, l'existence de plusieurs solutions ORB et la différence entre eux fait que l'outil de traçage requiert des implémentations spécifiques pour chaque ORB. L'échantillonnage se fait à de longs intervalles, ce qui s'avère imprécis pour trouver les goulots d'étranglement et les blocages d'événements au niveau des millisecondes. *NetLogger* [38] s'avère un outil d'analyse de performance efficace. Il a été utilisé dans des projets d'applications réparties de grande ampleur comme EUDataGrid, GLOBUS, GRID 3, Network Weather Service. Pour être employé dans tous les environnements d'applications réparties d'aujourd'hui, s'impose la nécessité d'ajouter d'autres supports que pour C/C++, Java et Perl. Le système de surveillance proposé par NetLogger implique comme hypotèse de travail l'accès au code source du projet.

Outre JavaRMI (qui est un modèle client-serveur) et IIOP de CORBA, d'autres solutions sont conçues pour l'invocation à distance: DIME [44], ICE [16] et Web-

Services [42] (qui ne se présentent pas comme une veritable application répartie [67]).

Aucun de ces outils ne se prête à l'analyse d'une application répartie qui utilise la nouvelle technologie *.Net Remoting*. La nouvelle architecture *.Net/Mono* et le protocole sous-jacent *.Net Remoting* permettent d'aborder d'un autre point de vue les interactions entre les processus d'une application répartie, une fois que l'état d'un tel processus (*AppDomain*) devient accessible par un autre. Les spécificités de cette nouvelle architecture sont discutées au prochain chapitre.

CHAPITRE 3

CONCEPTS DE BASE

3.1 Notions introductives CLI

Les applications réelles ont besoin d'une intégration facile à la plate-forme. Le cadre d'applications .Net/Mono [21, 68] offre une telle solution: une plate-forme pour la construction des applications en fournissant une infrastructure généralement décrite comme *intergiciel*. Le cadre se compose d'un environnement d'exécution CLI [48] (Common Language Infrastructure) et d'un ensemble de bibliothèques de classes, qui fournit une plate-forme de développement pouvant être exploitée par une variété de langages et d'outils.

CLI est un environnement dans lequel s'exécutent les applications gérées (gestion automatique de la mémoire), qui fournit une couche de fonctionnement entre les applications et le système d'exploitation. Le CLI est semblable à l'environnement d'exécution des langages interprétés comme Smalltalk ou la machine virtuelle Java. Cependant le CLI n'est pas un *interpréteur*. La différence essentielle est que le CLI n'interprète jamais la représentation IL (*intermediate language*) du langage; au lieu de cela il compile toujours l'IL dans le code natif, qui s'exécute directement sur le système. Parmi les services principaux, le CLI gère la mémoire, les fils d'exécution, l'exécution du code, la vérification de la sécurité du code et la compilation.

3.1.1 La sécurité d'application et l'intégration inter-langage

Le CLI fournit des spécifications pour le code exécutable CTS (Common Type System) et l'environnement d'exécution VES (Virtual Execution System [48]) dans lesquels il fonctionne. Il simplifie l'intégration inter-langage par l'introduction du système de type commun - CTS. Le CTS définit tous les types de base qui peuvent

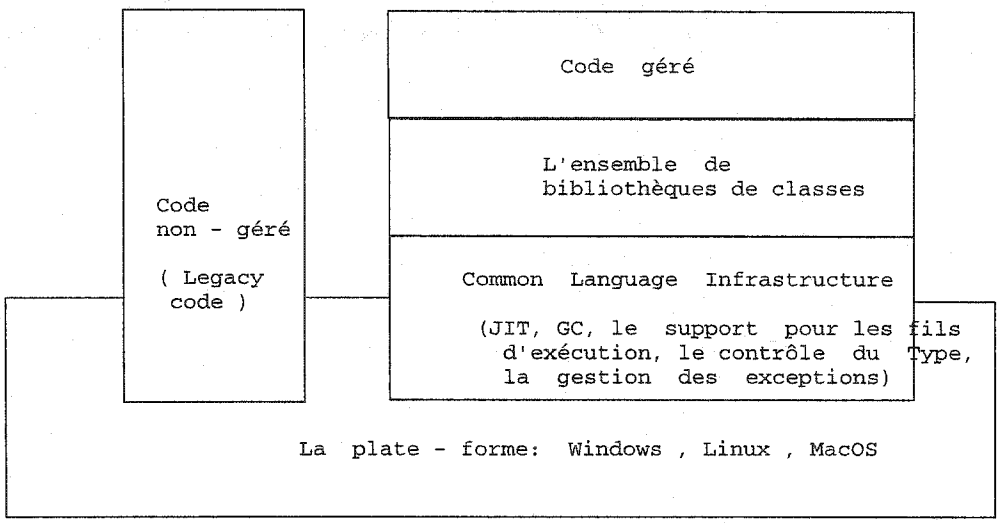

Figure 3.1 Le système global

être utilisés dans l'environnement et les opérations qui peuvent être exécutées sur ces types. Les applications peuvent créer des types plus complexes, mais ils doivent être construits à partir de types définis par le CTS. Tous les types CTS sont des classes qui dérivent d'une classe de base appelée *System.Object* (même les types *primitifs* tels que les entiers). CTS est un ensemble de conventions décrivant la façon dont les types sont utilisés et déclarés:

- *types valeur*: la mémoire est allouée sur une pile de type LIFO et chaque valeur occupe un nombre prédéfini d'octets de mémoire. Les objets de type valeur ont deux représentations: *unboxed* et *boxed* (i.e., directement ou par pointeur)

- *types référence*: utilisent la pile, mais la pile contient seulement l'adresse du prochain emplacement libre sur le tas.

Le CLI gère les définitions de type au moment de l'exécution. Les compilateurs de langages ciblant le CLI font la correspondance des abstractions du langage aux

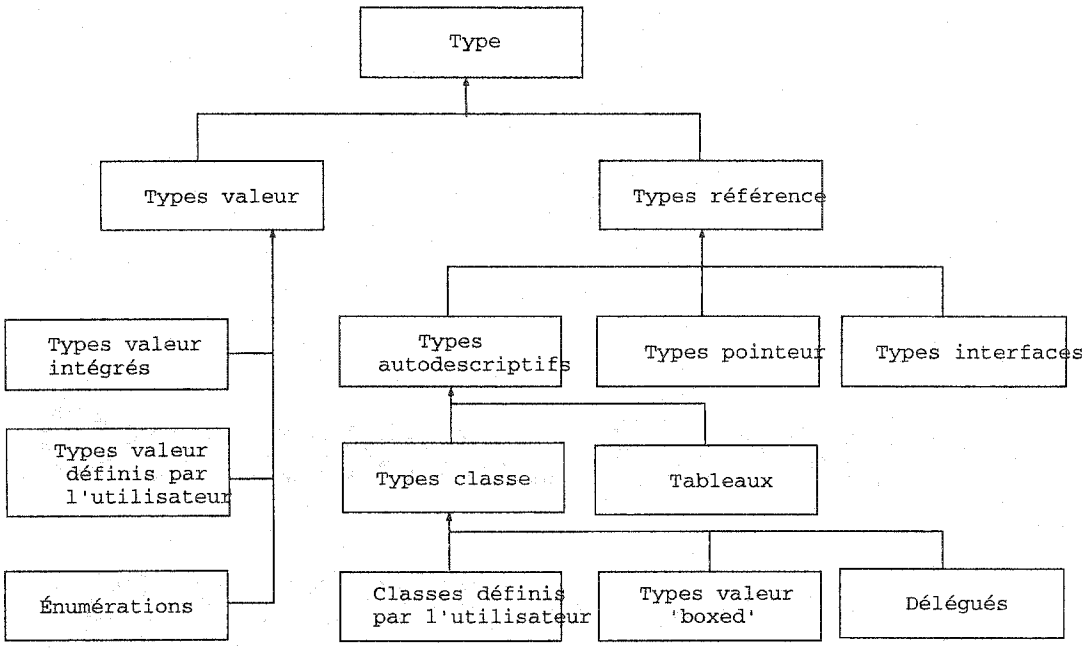

Figure 3.2 L' hiérarchie des types

définitions de type de CLI.

Le VES implémente et impose le modèle de CTS (fig. 3.2). Le VES est responsable du chargement et de l'exécution des programmes écrits pour CLI. Il fournit les services requis pour exécuter le code et les données gérées, en utilisant les métadonnées au moment de l'exécution pour relier les modules produits séparément.

L'environnement VES fournit le soutien direct pour un ensemble de types prédéfinis. Il définit une machine virtuelle avec un état et un modèle de machine associé, et un modèle de gestion des exceptions. Le CLI pour la plate-forme Windows s'appelle CLR alors que l'environnement Mono il s'appelle Mono CLI.

Le *code géré* est le code représenté comme CIL (Common Intermediate Language) [50]; son exécution est contrôlée par le VES. Le *code non-géré* est le code natif de la plate-forme d'exécution.

Les *données gérées* sont les données contrôlées par le ramasse–miettes du CLI. Les

données non-gérées sont allouées et relâchées manuellement. Dans certaines situations, le code géré doit interagir avec le code non-géré. Cette possibilité est offerte par différents mecanismes [56]

- l'appel de la plate-forme: P/Invoke (*Platform Invoke*) est un service qui permet au code géré d'appeler des fonctions non-gérées implémentées dans des bibliothèques de liaison dynamique comme celles figurant dans l'interface API (*.dll pour la plate-forme Windows ou *.so pour la plate-forme Unix);

- l'exposition de composants COM, sous Windows au .Net Framework (en utilisant l'outil *tlbimp.exe*);

- l'exposition de composants .Net Framework à COM, sous Windows (en utilisant les outils *tlbexp.exe/regasm.exe*)

Parce que l'environnement d'exécution est multi-langage, il soutient une large variété de types de données et différentes fonctionnalités. Pour faciliter l'interopérabilité inter-langages, les spécifications de langage commun définissent un sous-ensemble du CTS. Les types qui ne sont pas soutenus par le CLS (Common Language System) [48] peuvent être employés à l'intérieur par les classes exposées à d'autres langages, mais ne devraient pas être exposées comme champs publics ou comme paramètres ou valeurs de retour aux méthodes publiques. CLS représente un ensemble de règles qui doivent être suivies par une application conforme aux CLS. CLS constitue le plus petit dénominateur commun entre tous les langages. CLS définit un ensemble strict de 41 règles.
Par exemple la règle 1 dit que ces règles s'appliquent seulement aux parties d'un type accessible ou visible de l'extérieur de l'assemblage le définissant.

Le système d'exploitation crée un espace d'adressage de processus pour exécuter un programme. Le CLI doit fonctionner sous un processus du système d'exploitation dans un *espace d'adressage standard.* Lorsqu'il faut exécuter plusieurs

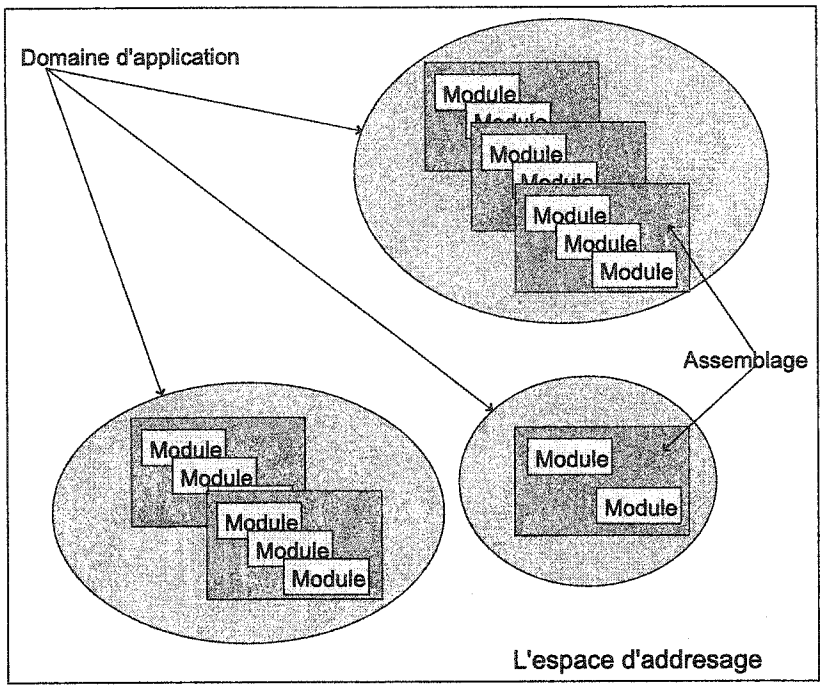

Figure 3.3 L'espace d'adressage, domaines d'application, assemblages et modules

petits programmes à l'intérieur du même processus, le VES crée une version légère d'un processus appelé *domaine d'application* (AppDomain).

L'unité créée par le compilateur est un *module* et les modules sont combinés pour former une unité déployable appelée *assemblage* (*assembly*).

Les assemblages peuvent s'exécuter par eux-mêmes, ou se combiner pour s'exécuter dans un domaine d'application. Plusieurs domaines d'applications peuvent s'exécuter dans un espace d'adressage simple, sans s'inquiéter de l'interférence, parce que le code contenu est sécurisé, ce qui signifie qu'il ne doit accéder qu'aux emplacements de mémoire autorisés.

Le système d'exploitation assure que de multiples espaces de mémoires puissent être supportés simultanément (fig. 3.3) en utilisant son propre mécanisme de protection de mémoire orienté-processus.

Un assemblage est stocké dans un fichier en format PE (*portable executable*) une extension du format COFF (Common Object File Format) utilisé traditionnelle-

ment pour le contenu exécutable. Ce format de fichier, qui accepte le code IL ou le code natif ainsi que les métadonnées, permet au système d'exploitation de reconnaître les images du CLI.

Quand le fichier contient un assemblage, le moteur d'exécution du CLI est chargé par le système d'exploitation et commence à s'exécuter. Après, le CLI contrôle le chargement et l'exécution de l'assemblage comme code géré. L'assemblage est chargé dans un domaine d'application défini par le CLI au démarrage. D'une manière abstraite, chaque domaine d'application définit un espace d'adressage contrôlé par le moteur d'exécution du CLI, plutôt que par le système d'exploitation. Toutes les références d'adresses sont vérifiées (du point de vue de leur type) pour s'assurer qu'elles sont à l'intérieur du domaine d'application.

Une fois que le code est écrit, le compilateur le traduit dans un langage intermédiaire IL (une forme abstraite et intermédiaire, indépendante du langage de programmation initial, de la machine cible et de son système d'exploitation).

À la suite de la représentation dans cette forme abstraite, les applications écrites en différents langages peuvent interagir étroitement, non seulement au niveau d'appel des méthodes de l'autre, mais également au niveau de l'héritage de classe.

La représentation intermédiaire (IL) des applications gérées, destinée à l'environnement CLI, inclut deux composants principaux: *les métadonnées* et *le code géré*. Au moment de l'exécution, le CLI n'a aucune idée du langage de programmation qui a été employé par le développeur. Ceci signifie qu'on devrait choisir n'importe quel langage de programmation qui permette d'exprimer nos intentions le plus facilement possible, tant que le compilateur utilisé cible le CLI.

Les caractéristiques du CLI sont disponibles à tous les langages de programmation qui le visent (C#, VB, C++, J#, NetCobol). Si des exceptions sont employées pour rapporter des erreurs, ces exceptions pourront être traitées de n'importe quel langage. Le code CIL est parfois mentionné comme code contrôlé (ou protégé) parce que le CLI contrôle son cycle de vie et son exécution. De plus, chaque compi-

lateur ciblant le CLI doit produire des métadonnées complètes pour chaque module géré.

3.1.2 Les métadonnées

Les métadonnées [49] représentent un ensemble de tables de données qui décrivent le nom du module ou de l'assemblage, les noms de tous ses composants, leur types et les noms de toutes les fonctions membres, y compris les noms et types de paramètres. Avec ces informations, l'environnement d'exécution est capable de créer des objets, d'appeler les fonctions, ou d'accéder aux données des objets. Les compilateurs peuvent aussi les employer pour trouver quels objets sont disponibles et comment un objet est utilisé.

Les métadonnées ont des tables qui indiquent les références aux autres modules gérés, telles que les types importés et leurs membres. Les métadonnées représentent un surensemble des anciennes technologies telles que le catalogue de types (COM) et les fichiers du langage de description (IDL). La chose la plus importante à noter est que les métadonnées générées par CLI sont plus complètes et, à la différence de leurs ancêtres, elles sont toujours associées au fichier qui contient le code IL (EXE ou DLL).

Puisque le compilateur produit les métadonnées et le code IL en même temps et les lie dans le module géré, ces deux éléments ne sont jamais désynchronisés.

Structurellement, les métadonnées sont comme une base de données relationnelle normalisée; les métadonnées sont organisées sous forme d'un ensemble de tables de renvois, par opposition à une base de données hiérarchique qui a une structure arborescente. Chaque colonne de chaque ligne d'un tableau de métadonnées contient des données ou une référence à une ligne d'un autre tableau. Les métadonnées ne contiennent aucun champ d'information double. Chaque catégorie de données réside dans une seule table de la base de métadonnées. Si une autre table doit utiliser les mêmes données, elle réfère à la table qui contient les données.

3.1.3 L'assemblage

Le CLI ne fonctionne pas avec des modules, mais avec des *assemblages*. Un assemblage est un concept abstrait: il est d'abord un groupement logique d'un ou plusieurs modules ou fichiers de ressource (bmp, jpg,..). Il est l'*unité fondamentale* de déploiement et d'activation, ce qui permet le contrôle de version et la réutilisation.

Selon les choix faits avec les compilateurs ou les outils, on peut produire un assemblage d'un seul fichier ou un assemblage multi-fichiers(fig. 3.4). L'assemblage permet de découpler les notions logiques et physiques d'un composant réutilisable, la répartition de code et des ressources dans différents fichiers étant complètement paramétrable.
Les modules d'un assemblage incluent des informations, comme le numéro de version sur les assemblages référencés; les informations rendent l'assemblage auto descriptif. Cela signifie que le CLI connaît tout ce dont l'assemblage a besoin pour s'exécuter. Les caractéristiques d'un assemblage sont les suivantes:

- il offre une *image commune du code*: tous les compilateurs ciblant le CLI produisent des assemblages;
- il emploie un *langage neutre*: n'importe quel langage peut produire l'assemblage, seulement des notions de CLS sont utilisées;
- il est *indépendant de plate-forme*: l'assemblage contient seulement le code abstrait (code IL);
- il est *auto-descriptif*: il contient toutes les informations sur lui-même (code et métadonnées)

Le format d'un assemblage peut contenir les instructions spécifiques de l' UCT cible pour éviter la compilation du JIT au moment de l'exécution et pour permettre l'optimisation pour l'UCT ciblée.

La capacité d'auto-description des métadonnées devient plus évidente quand on considère le processus de la *réflexivité*, qui permet à un développeur d'interroger et d'utiliser directement les métadonnées d'un assemblage. Il est possible de lister toutes les classes contenues par un assemblage, les membres exposés par les classes, et les paramètres prévus par les membres, sans connaissance antérieure de l'assemblage. En utilisant cette information, il est possible d'appeler dynamiquement la méthode d'une classe, en construisant ses paramètres en cours d'exécution. Vu de l'extérieur, un assemblage est une collection de types exposés, qui pourraient être les types de base trouvés dans le CTS comme des entiers et des chaînes de caractères, ou des constructions plus complexes, tels que des classes, des structures et des énumérations.
Rappelons qu'un domaine d'application peut contenir de multiples assemblages et qu'un assemblage peut contenir de multiples modules. Une telle hiérarchie est montrée dans la figure 3.5.

La réflexivité est typiquement utilisée pour les bibliothèques des classes qui doivent comprendre la définition d'un type afin de fournir une fonctionnalité particulière. Par exemple, le mécanisme de sérialisation utilise la *réflexivité* pour déterminer quels champs définissent le type. Le formateur de sérialisation (binaire ou de type XML) peut obtenir les valeurs de ces champs et les écrire dans un flux d'octets pour les envoyer à travers le réseau (local ou Internet).
La réflexivité est utilisée quand une application doit charger un type spécifique d'un certain assemblage à l'exécution pour accomplir une certaine tâche.
En général, l'utilisation de la réflexivité pour appeler une méthode ou pour accéder à un champ ou à une propriété est plus lente, pour plusieurs raisons:

- le processus de liaison cause beaucoup de comparaisons de chaîne de caractères durant la recherche du membre désiré.

- le passage des arguments exige qu'un tableau soit construit et que les éléments

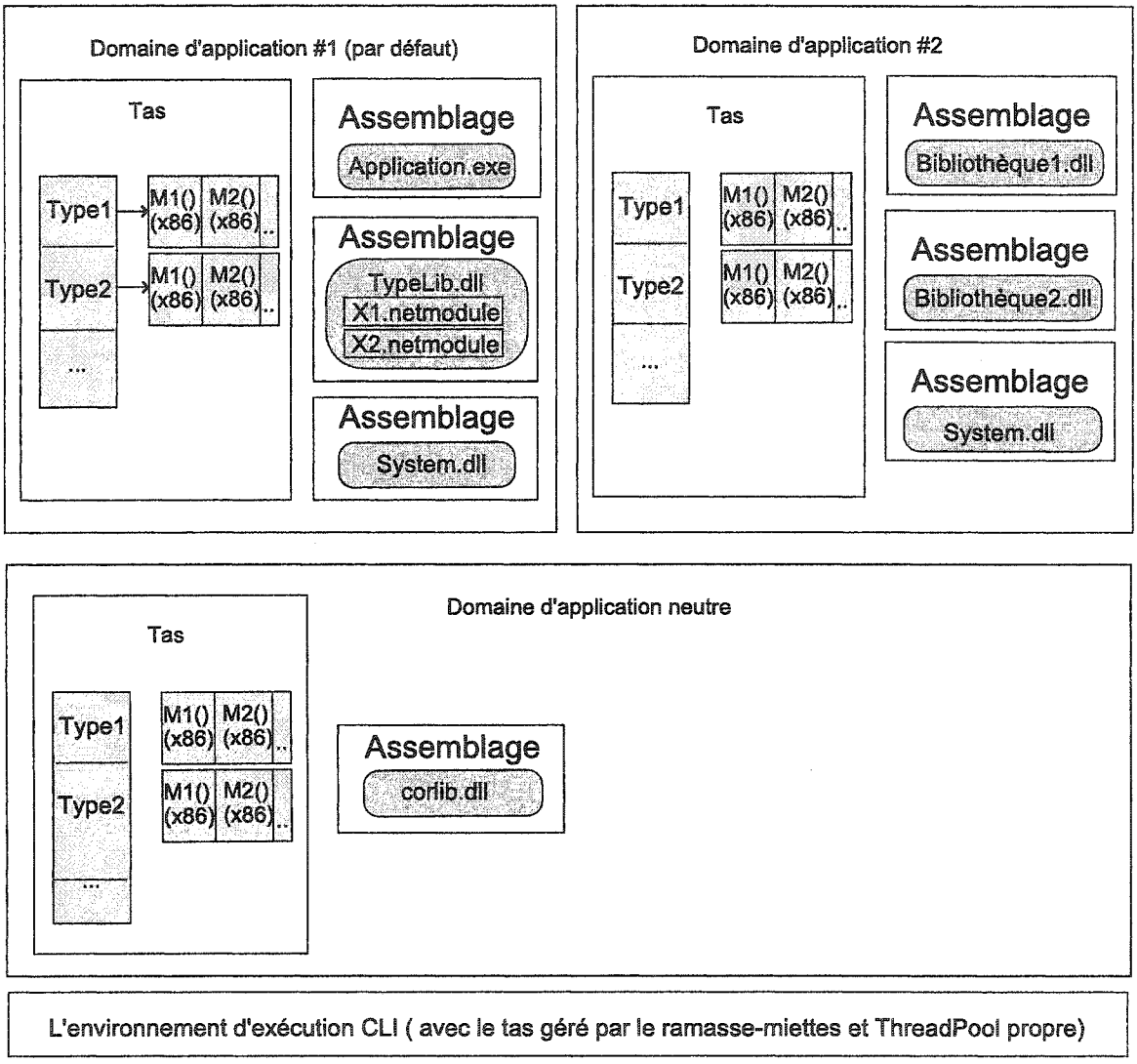

Figure 3.4 Une vue d'un processus avec CLI et deux domaines d'application. À chaque Type corespond un ensemble de métadonnées [65]

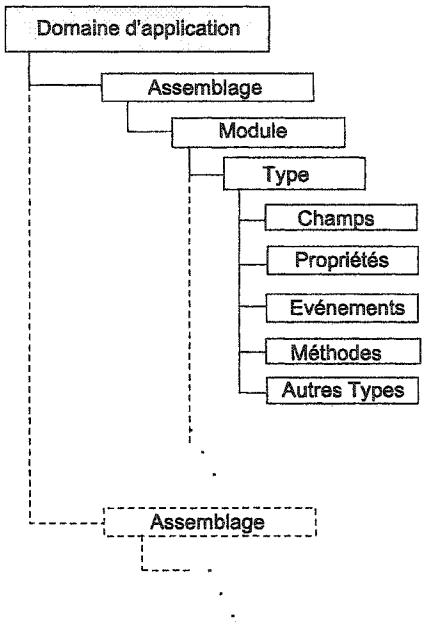

Figure 3.5 Les composants d'un domaine d'application

du tableau soient initialisés. L'appel d'une méthode exige que les arguments soient extraits et placés sur la pile.

- le CLI doit vérifier que les nombres et types de paramètres passés à une méthode sont corrects.

- le CLI vérifie dynamiquement que l'appelant a la permission d'accéder au membre.

La réflexivité est un outil puissant pour des développeurs. *La réflexivité* permet de concevoir des applications extensibles dynamiquement.

3.1.4 Les fils d'exécution

Le CLI a comme tâche la gestion de fils d'exécution (*threads*). Il supporte l'exécution indépendante de différents objets en utilisant la notion du *fil d'exécution CLI* ou *fil d'exécution logique* (du point de vue sémantique c'est un concept distinct

d'un fil d'exécution du système d'exploitation). Le CLI maintient sa propre réserve de processus légers, une collection d'objets dont chacun se comporte comme une unité indépendante de calcul. La réutilisation des fils implique une pénalisation du système plus faible qu'en cas de production des nouveaux fils.

La classe *System.Threading.ThreadPool* fournit cette réserve qui permet la distribution facile du travail parmi plusieurs fils d'exécution. Elle constitue une construction idéale pour traiter les tâches multiples de courte durée qui peuvent être accomplies en parallèle (comme la gestion des E/S asynchrones, l'exécution asynchrone de délégations, la gestion des minuteries, la communication par interfaces de connexion).

Les processus légers appartenant à cette classe ne peuvent pas être contrôlés comme les processus légers de type logique. Le tableau montre une comparaison entre les deux types de fils d'exécution [39]. Le cadre d'application différencie entre

Tableau 3.1 Comparaison entre les fils d'exécution CLI

	ThreadPool	*GenericThread*
Idéal pour les tâches courtes	Oui	Non
Contrôle du nom	Non	Oui
Contrôle de la priorité	Non	Oui
Contrôle du cycle de vie	Non	Oui
Flexibilité	Non	Oui
Contrôle de synchronisation	Non	Oui

les *fils d'exécution physiques* (contrôlés par le système d'exploitation) et des *fils d'exécution logiques* (gérés par CLI et qui fournissent au delà d'un processus léger physique) des fonctionnalités supplémentaires. L'implémentation actuelle de CLI utilise une correspondance de type 1 : 1 entre les deux types de fils d'exécution.

3.1.5 JIT

L'environnement CLI fournit le service de compilation JIT, pour convertir le code intermédiaire IL en code natif. Ce compilateur est appelé typiquement par le

moteur d'exécution au premier appel de la méthode (on utilise cette facilité dans la présente étude pour trouver le temps passé dans un appel de type .Net Remoting). La compilation JIT (fig. 3.6) se produit au fur et à mesure des appels de méthodes. Le code peut être pré-compilé en code natif pendant le déploiement. Le JIT présente l'avantage de s'exécuter sur la machine où sera exécuté le code. Ceci confère la possibilité d'optimiser le code, spécifiquement pour le système où il s'exécute. Le plus grand avantage du langage IL n'est pas qu'il fasse abstraction

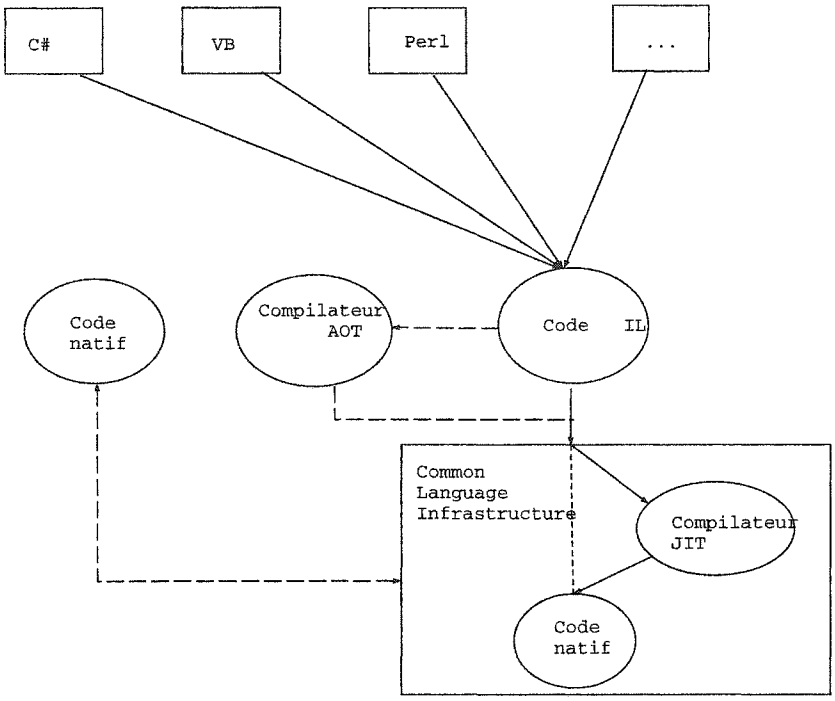

Figure 3.6 Modèle d'éxecution

de l'UCT, mais qu'il permette de vérifier la sécurité. Pendant la compilation de IL vers des instructions natives, le CLI examine le code IL de haut niveau et s'assure que tout ce qu'il fait est sécuritaire (e.g. aucune case mémoire n'est lue avant d'être écrite, chaque méthode est appelée avec le nombre correct de paramètres et chaque paramètre est du type correct, la valeur de retour de chaque méthode est utilisée

correctement, chaque méthode a une instruction de retour).
Quand le code est de type sécurisé, le CLI peut complètement isoler des assemblages les uns des autres même s'ils s'exécutent dans la même processus. Cet isolement aide à s'assurer que les assemblages ne peuvent pas se compromettre et augmente la fiabilité d'application.
Pendant la vérification, certaines parties du code de type sécurisé peuvent ne pas passer le test avec succès en raison des limitations du processus de vérification. De plus, ce ne sont pas tous les langages qui produisent un code de type sécurisé vérifiable (C# est vérifiable alors que C et C++ ne le sont pas).
Les métadonnées du module géré incluent tous les informations du type et des méthodes employées par le processus de vérification. Si le code IL est considéré comme dangereux, une exception est signalée au moment du chargement empêchant l'exécution de la méthode.

3.1.6 Gestion de la mémoire

Toutes les ressources des applications gérées sont allouées par un tas, en utilisant un ramasse-miettes(GC) de type 'Mark and Compact'. Ce segment de mémoire est semblable à un tas pour le langage C, sauf que les objets sont automatiquement libérés quand l'application n'a plus besoin d'eux.
Pour le tas géré, l'allocation d'un objet représente une simple addition d'une valeur à un pointeur; c'est bien plus rapide qu'une allocation de mémoire en C.
En général, les trois hypothèses suivantes sont vérifiées:

- plus les objets sont nouveaux, plus courte est leur vie,
- plus les objets sont anciens, plus longue est leur vie,
- en collectant seulement une partie du tas, un gain en temps est obtenu.

Ainsi, le GC du CLI sépare le tas géré en trois générations (0, 1 et 2). Quand l'application commence, le tas est vide. Une fois que des objets sont ajoutés au

tas, ils appartiennent à la génération 0.

Une fois que la taille de tous les objets de la génération 0 surpasse l'espace assigné, le ramasse–miettes s'exécute et recupère tous les objets non-utilisés de la génération 0. Tous les objets qui survivent deviennent des objets de génération 1. Les mêmes étapes sont répétées pour la génération 2. La collecte d'objets non-utilisés se produit seulement quand la génération 0 est pleine; entre temps, le tas géré est sensiblement plus rapide qu'un tas en langage C.

Dans la hiérarchie d'espace d'adressage du CLI, il y a théoriquement une autre couche: les domaines d'application sont chargés dans *l'espace d'adressage de CLI* [59].

Un espace d'adressage du CLI correspond à un espace d'adressage de processus (l'espace d'adressage du CLI est distinct de l'espace d'adressage du SO de sorte que la sémantique du CLI est distincte de n'importe quelle définition de l'espace d'adressage d'un système d'exploitation particulier).

Chaque espace d'adressage peut contenir de multiples domaines d'applications. Il y a un domaine d'application pour les assemblages du système, et d'autres qui contiennent les assemblages qui peuvent être partagés avec d'autres domaines d'applications. Le mécanisme d'interaction entre les espaces de mémoire des domaines d'application bloquent l'interaction entre les domaines d'application.

Le CLI supporte la communication entre les domaines d'application (fig. 3.7) par le biais d'un mécanisme spécifique, appelé *Remoting*. Ce même mécanisme peut également être employé pour communiquer entre des processus distants. Le *Remoting* est en effet le mécanisme qui permet la communication entre le code d'un domaine d'application avec des types et des objets contenus dans un autre domaine d'application.

La plupart des types supportent une *conversion des paramètres par valeur* à travers les frontières du domaine d'application. Le CLI doit sérialiser les champs de l'objet dans un bloc de mémoire. Le bloc passe alors à une autre domaine d'application,

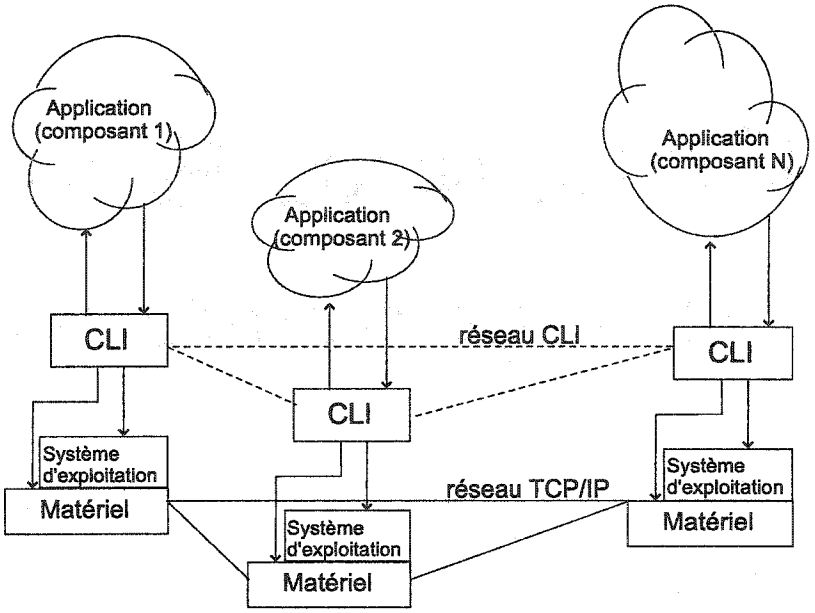

Figure 3.7 L'infrastructure CLI répartie

qui le désérialise pour produire un nouvel objet. Le domaine d'application destination n'a *aucun* accès à l'objet du domaine d'application initial.

Cependant, beaucoup d'objets ne peuvent pas ou ne devraient pas être copiés et déplacés vers un autre processus pour l'exécution (par exemple les objets extrêmement larges, avec beaucoup de méthodes). Habituellement, un client a besoin seulement de l'information retournée par une ou quelques méthodes sur l'objet-serveur. Par conséquent, copier l'objet-serveur entièrement serait une perte de largeur de bande aussi bien que de mémoire du client et du temps machine. Dans de telles situations, le processus serveur devrait passer au processus client une référence à l'objet de serveur, et non pas une copie de l'objet.

Les types qui sont dérivés du type *System.MarshalByRefObject* peuvent également être accédés à travers des frontières d'un domaine d'application. Cela est possible car l'accès à l'objet-serveur est accompli par *référence* plutôt que par valeur.

Si un domaine d'application contient un objet créé, dont le type dérive de *MarshalByRefObject*, et si une référence à ce type est passée à un domaine destination,

le CLI crée une instance de type mandataire (*Proxy*) dans le domaine destination. L'objet initial et ses champs demeurent dans le domaine d'application initial. L'*objet mandataire* est une classe d'encapsulation qui sait comment appeler des méthodes sur l'objet-serveur dans le domaine d'application initial, sans que la destination ait un accès direct à l'objet du domaine d'application initial (l'objet-serveur a une identité réseau).
Cela permet différentes modalités d'interaction entre le client et le serveur, y compris la capacité pour un objet-client d'appeller de façon asynchrone un objet distant, recevant le résultat ou une exception en retour, avec divers paradigmes de type IPC.

Local ou Distant ?

En principe, tous les objets appartenant au même domaine d'application sont *locaux*. Les appels de méthodes pour un objet local s'exécutent immédiatement via la pile d'appel, sans qu'une conversion de paramètres ne se produise.
Tous les autres objets sont considérés *distants*, même si les domaines d'applications appartiennent au même processus. Les appels de méthodes (y compris champs, propriétés, événements délégués et méthodes exposées par la classe) pour un objet distant s'exécutent via l'objet mandataire et utilisent le protocole *.Net Remoting*, tous les arguments étant soumis à une conversion de paramètres.

3.2 .Net Remoting

Le *Remoting* permet de réaliser une application composée de briques logicielles (composants) réparties sur plusieurs machines. Ces composants peuvent s'appeler de manière transparente par rapport à leur emplacement.

L'architecture du *.Net Remoting* (fig. 3.8) est basée sur cinq éléments:

- L'*objet mandataire*: la représentation locale d'un objet distant qui achemine les appels effectués sur l'objet de type *Proxy* vers ce dernier.
- *Messages*: des objets qui contiennent les données nécessaires pour l'exécution d'un appel de méthode à distance.
- *Récepteurs de message*: des récepteurs fournis par l'infrastructure distante qui permettent le traitement approprié de l'appel de méthode distante.
- *Formateurs*: des récepteurs qui fournissent les fonctionnalités de base (sérialisation et désérialisation des objets) pour les formats de transfert.
- *Récepteurs de canal*: transfèrent tous les messages sérialisés envoyés vers ou en provenance d'un domaine d'application.

3.2.1 Les messages et les composants d'objet mandataire

Pour communiquer avec des objets distants, l'infrastructure *.Net Remoting* utilise des messages qui sont employés pour transmettre des appels de méthode distants, pour activer des objets distants et pour communiquer des informations. Un message est seulement un objet dictionnaire (*IDictionary*) caché derrière l'interface *IMessage*. Même si tous les messages sont basés sur cette interface, le cadre d'application définit plusieurs types (fig. 3.9): *ConstructionCall, MethodCall*. La différence entre ces types de messages est la prédéfinition de certaines entrées dans le dictionnaire interne.

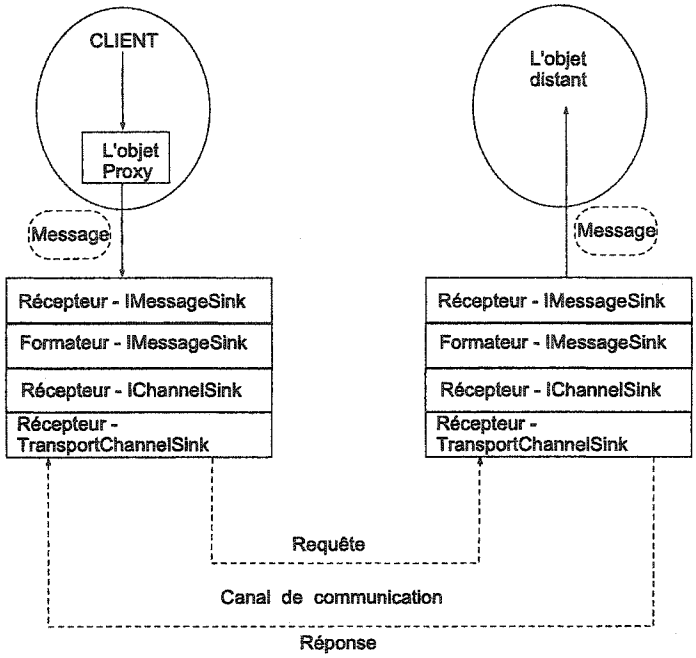

Figure 3.8 *.Net Remoting* - vue d'ensemble

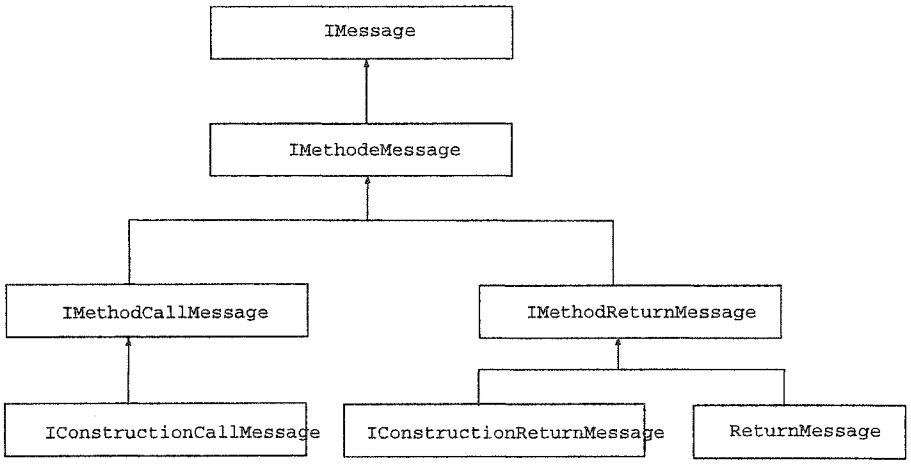

Figure 3.9 Les types de messages

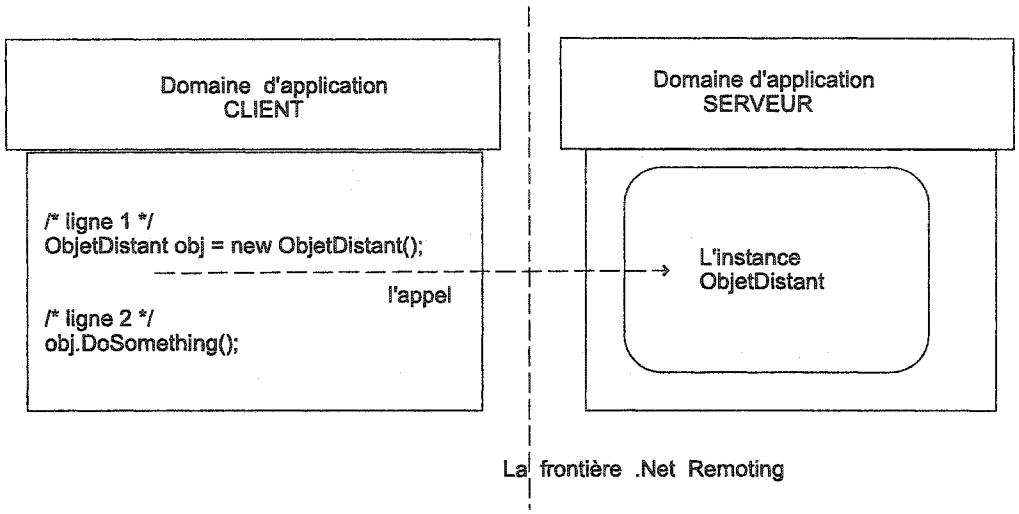

Figure 3.10 La vue du développeur

Le dictionnaire représente des collections de paires clé–valeur. Il est possible d'y ajouter n'importe quel type d'objet, mais les objets utilisés doivent être sérialisables afin d'être transportés à travers une frontière distante. L'instance *obj* de l'objet distant (ligne 1 - fig.3.10) est convertie dans l'instance d'un message .Net Remoting (*ConstructionCallMessage*).

Un message qui passe entre le client et le serveur est généré par l'objet *RealProxy*. En utilisant l'opérateur *new*, ou en appelant *Activator.GetObject()* pour acquérir une référence à un objet distant, le *Remoting* produit deux objets mandataires: *TransparentProxy* et *RealProxy*.

La requête d'activation d'un objet retourne une instance de *ObjRef* (référence d'objet) et celui-ci est placé dans un objet *TransparentProxy*. *ObjRef* est une structure sérialisable avec l'information nécessaire pour la production d'un objet mandataire. Il contient toute l'information exigée pour localiser et accéder un objet distant. L'utilisation des références d'objets pour communiquer entre les objets-serveurs et les clients est le cœur du *Remoting*. L'infrastructure emploie des objets

mandataires pour créer l'impression que l'objet-serveur est dans le processus client. En utilisant l'activation *Activator.GetObject()*, ou l'opérateur *new*, il n'y a aucun échange entre les deux entités (client et serveur) jusqu'au premier appel de méthode.

L'objet mandataire est créé sur le client à partir de métadonnées, et le serveur active l'objet au premier appel. En réalité, au moins trois objets de type mandataire font partie d'un appel *Remoting*: *TransparentProxy*, *RealProxy* et le descendant - *RealProxy (RemotingProxy)* [2, 3].

Après la ligne 1 (fig. 3.10), *obj* réfère vraiment à un *TransparentProxy*, qui est fourni par le moteur d'exécution, en tant qu'implémentation des caractéristiques grâce au code IL. Cet objet *TransparentProxy* a une référence au *RealProxy*. Le *TransparentProxy* représente l'ObjetDistant, ce qui permettra à toutes les méthodes de ObjetDistant d'être appelées par cet intermédiaire.

Quand la ligne 2 (fig. 3.10) est exécutée, le *TransparentProxy* produit un objet *MessageData* qui contient quelques pointeurs internes et appelle une fonction privée de *RealProxy (PrivateInvoke())*, qui va créer un nouvel objet (*IMesage*).

L'objet *IMessage* charge maintenant son dictionnaire interne avec les pointeurs du *MessageData*. Après cela, l'appel sera transféré à la méthode *RemotingProxy:Invoke()*. Les raisons pour lesquelles les trois proxies sont nécessaires sont les suivantes:

- Le *TransparentProxy* peut se déguiser en n'importe quel autre objet et peut produire les données nécessaires au message;
- Le *RealProxy* génère vraiment l'objet *IMessage*;
- Le descendant de *RealProxy* va traiter le message.

TransparentProxy et *RealProxy* sont créés de manière interne lorsque l'objet est activé, mais seul le *TransparentProxy* est renvoyé au client par la méthode *GetTransparentProxy()* de l'objet *RealProxy*.

La classe *TransparentProxy* est une classe interne qui ne peut pas être enrichie ou étendue. C'est la classe *RealProxy* qui est soumise à de tels effets, dans le cas où l'on désire implémenter les fonctionnalités suivantes:

- La répartition de charge: il suffit de tester si tel ou tel serveur est chargé dans le but de répartir efficacement les traitements.
- L'interception des paramètres pour ajouter certaines informations (traduction linguistique des chaînes de caractères en temps réel, par exemple).
- Étendre les fonctionnalités de sécurité existantes pour seulement acheminer les appels vers les serveurs auprès desquels le client est authentifié.

L'objet mandataire présente une double nature: il joue le rôle d'un objet de la même classe que l'objet distant (l'objet mandataire transparent) et il est lui-même un objet géré.

3.2.2 Les récepteurs IMessageSink

Les appels de méthode à distance sont définis en termes de messages (*IMessage*) qui sont envoyés du client au serveur (requête) éventuellement avec retour. Le transfert d'un message à partir d'une application client à un objet situé du côté-serveur passe par une chaîne d'objets *IMessageSink*. Les objets *IMessageSink* sont enchaînés dans le sens que chaque *IMessageSink* est responsable pour l'appel de la méthode *ProcessMessage()* sur le prochain *IMessageSink*, après qu'il ait effectué son travail.

Un récepteur recevra un message d'un autre objet, appliquera son propre traitement, et déléguera n'importe quel travail additionnel au prochain récepteur de la chaîne. Les récepteurs de messages et les contextes sont utilisés pour empêcher un client d'appeler une méthode à distance en passant des paramètres invalides à une méthode, évitant ainsi un aller-retour à l'objet distant. Les récepteurs peuvent être

utilisés aussi pour tracer les messages et les exceptions levées à travers les frontières du contexte ou du domaine d'application.

Un objet message passe d'un récepteur à l'autre dans la chaîne et transmet une série de propriétés nommées telles que des identificateurs d'action, des informations d'envoi et des paramètres. Chaque type qui implémente l'interface *IMessageSink* va participer à l'infrastructure *Remoting* comme un récepteur. Les membres de l'interface *IMessageSink* sont:

- *NextSink*: propriété publique qui réfère au récepteur de message suivant.

- *IMessageSink.SyncProcessMessage()*: méthode qui traite la requête d'une façon synchrone en passant ce message à la méthode *SyncProcessMessage* du prochain récepteur de la chaîne. Le traitement synchrone s'achève après que l'infrastructure Remoting reçoive et retourne le message réponse de la méthode appelée, auquel cas la valeur de retour est un objet *IMessage* qui contient la valeur de retour de l'appel de méthode, et tous les paramètres de type *out*.

- *IMessageSink.AsyncProcessMessage()*: méthode qui traite l'appel distant d'une façon asynchrone. Ce type d'appel ne traite pas à la fois les deux messages: requête et réponse. *AsyncProcessMessage()* traite seulement la requête et continue l'exécution (non bloquante). Dans le traitement asynchrone, le retour n'est plus relié à l'appel, un récepteur doit être défini pour recevoir la réponse. Le message de réponse sera traité de façon synchrone, mais la génération du message se produit d'une manière asynchrone du côté–serveur.

L'objet mandataire contient une référence au premier *IMessageSink* qu'il doit employer, pour commencer la chaîne. Pour les appels asynchrones, au moment de la délégation (*BeginInvoke*), chaque récepteur désigne un récepteur de réponse (un autre *IMessageSink*) qui sera appelé par le prochain récepteur quand la réponse est

renvoyée. La nature de l'opération exécutée par un *IMessageSink*, quand il reçoit un objet *IMessage* pour le traiter, diffère d'un type de récepteur à l'autre.
Par exemple, un récepteur pourrait prendre un verrou, certains pourraient imposer des règles de sécurité. Deux ou plusieurs récepteurs dans la chaîne peuvent communiquer et interagir pour réaliser une tâche spécifique.

3.2.3 Les canaux

Un canal est le mécanisme de transport entre les domaines d'application et est responsable du transfert des messages. Les canaux sont des abstractions d'un protocole spécifique de transport et enveloppent le protocole pour le rendre disponible à l'infrastructure Remoting.
Un canal est responsable d'établir la communication de bout en bout. La partie transport de l'implémentation d'un canal distribue un flot de données (*data stream*) et est responsable de la réception du flot de données. Elle recevra aussi un flux de données d'un autre point final et est responsable de sa livraison à son domaine d'application.
Les canaux peuvent écouter et envoyer des messages. Tous les canaux sont dérivés de *IChannel* et implémentent *IChannelReceiver* (la fonction de lecture seule) ou *IChannelSender* (la fonction d'écriture seule), selon le but du canal. La plupart des canaux implémentent à la fois les deux interfaces pour permettre une communication bidirectionnelle.
Les canaux sont enregistrés par le domaine d'application. Les applications serveurs qui hébergent des objets répartis doivent enregistrer les canaux dont elles ont besoin. Lorsqu'un canal est enregistré, il démarre automatiquement l'attente des demandes de clients sur le port spécifié. Les récepteurs du canal sont responsables avec l'expédition de tous les appels reçus vers le prochain récepteur de la chaîne, et devraient fournir un mécanisme d'enregistrement de la référence du prochain récepteur (la propriété *NextChannelSink*).

Au début du dialogue client–serveur, chaque récepteur de canal implémente soit *IClientChannelSink*, soit *IServerChannelSink*, des interfaces produites par les fournisseurs de récepteurs de canal (*IClientChannelSinkProvider* et *IServerChannelSinkProvider*).

Le récepteur de canal *ClientChannel* est configuré selon le patron *Factory* [43] et le récepteur de canal *ServerChannel* demande une gestion de fils d'exécution appartenant à la classe *ThreadPool*.

Par défaut, l'infrastructure Remoting comprend deux types de canaux pour communication internet et intranet: *HttpChannel* et *TcpChannel*. Les canaux TCP codent des objets et des appels de méthode en format binaire pour la transmission en utilisant les interfaces de connexion. Les canaux HTTP codent des objets et des appels de méthode en format SOAP pour la transmission.

L'extensibilité de l'infrastructure *Remoting* permet de remplacer les récepteurs des canaux, en fonction des besoins de l'application, par d'autres types de canaux existants sur le marché: GenuineChannel [33], JabberChannel [46], DIME Channel [52], SMTP Channel [63].

3.2.4 Les récepteurs formateurs

L'infrastructure Remoting utilise des récepteurs de format pour convertir les objets *IMessage* vers un flux (*stream*) qui est passé aux récepteurs de canal pour l'envoi à l'objet distant. L'infrastructure Remoting sépare la fonctionnalité de mise en forme dans deux types: *récepteur de formateur client* et *récepteur de formateur serveur*. Les formateurs de sérialisation *IRemotingFormatter* sont basés sur l'implémentation de l'interface *IFormatter*.

Par défaut, deux types de formateurs sont proposés: *SoapFormatter* et *BinaryFormatter*, tous deux implémentent l'interface *IRemotingFormatter*.

Du côté-client, il y a l'interface *IClientFormatterSink* et, du côté-serveur, il y a les interfaces *IChannelSinkBase* et *IServerChannelSink*. *IClientFormatterSink* est

seulement une interface qui combine les interfaces *IChannelSinkBase, IClientChannelSink et IMessageSink*. Ce type de formateur assure la transition entre *IMessageSink* et *IClientChannelSink*; l'un travaille avec des messages, l'autre avec des flux.

Les formateurs sont insérés dynamiquement par l'architecture du canal. Les formateurs sont enfichables dans le canal utilisé. Par exemple le canal Http peut avoir plusieurs formateurs insérés (*XMLFormatter, BinaryFormatter, CustomFormatter..*). Les mêmes formateurs peuvent être branchés à un autre canal (par exemple *TcpChannel*).

Le cadre CLI permet de sérialiser les objets sous deux formes : un fichier XML ou un fichier binaire. La *sérialisation XML* ne permet de sauvegarder que les variables publiques et les propriétés, et ignore les méthodes, les champs privés, et les propriétés en lecture seulement. La *sérialisation binaire* permet d'enregistrer l'ensemble des éléments de l'objet.

Au sein de l'infrastructure, il existe deux façons de gérer la création des instances des composants:

- *CAO* (Client Activated Object): c'est le client qui gère la création de l'instance sur le serveur
- *SAO* (Server Activated Object): c'est le serveur qui est en charge de créer l'instance et qui la détient. À ce moment-là, il existe deux modes de fonctionnement au niveau des instances:
  - Mode *Singleton*: une seule instance est maintenue sur le serveur et partagée par tous les clients
  - Mode *SingleCall*: à chaque demande client (appel de méthode), le serveur crée une instance, réalise le traitement et détruit l'instance, libérant ainsi immédiatement les ressources du côté du serveur.

Figure 3.11 *.Net Remoting* - vue en détail [64]

En bref, en utilisant le protocole *.Net Remoting*, les deux joueurs (client et serveur) parcourent les étapes suivants:

Côté-client:

- Le client appelle la méthode sur l'objet TransparentProxy,
- TransparentProxy produit les données nécessaires au message (la conversion de la pile d'appel en MessageData) et les passe vers l'objet RealProxy,
- RealProxy produit un objet derivé de IMessage et l'expédie aux récepteurs de message,
- Le dernier récepteur de message expédie le message aux récepteurs de canal,
- Le premier récepteur de canal transforme le message en un flux d'octets,
- Le dernier récepteur de canal envoie le flux d'octets au serveur.

Côté-serveur:

- Le récepteur de canal reçoit un flux d'octets,
- Le récepteur expédie le flux à la chaîne de récepteurs de canal (le dernier récepteur de canal convertit le flux en message),
- Le dernier récepteur expédie le message à la chaîne de récepteurs serveur (le dernier récepteur du serveur est une pile d'appel),
- La pile d'appel émet l'appel à la méthode demandée.

Tous les composants de *.Net Remoting* sont paramétrisables à plusieurs niveaux, et ils peuvent utiliser différents protocoles et encodages. L'analyse de telles applications devrait aider à choisir les valeurs optimales pour ces paramètres. Toutes les implémentations existantes (.Net Framework, Mono, DOTGnu, Rotor), s'appuient sur des protocoles ouverts, (SOAP, DISCO, HTTP, XML, XSD, TCP) ce qui permet la réutilisation pour une productivité accrue.

CHAPITRE 4

LA PROPOSITION EXPÉRIMENTALE

L'optimisation des applications se fait à la suite d'une analyse statique et dynamique. Pour les applications réparties, c'est surtout la vue dynamique qui aide à comprendre le comportement du système et de chaque composant.

Pour l'analyse dynamique l'utilisation des classes *System.Diagnostics.Trace* ou *ITrackingService* permet d'obtenir certains événements mais qui ne suffisent pas; il manque des événements importants de l'infrastructure: sérialisation et desérialisation des messages. Aussi, le traçage via les attributs personnalisés qui s'ajoutent aux métadonnées de l'objet, méthode, type visé, en redefinissant le traitement pour les *MarshalByRefObject*, introduit des délais de traitement inacceptables.

L'environnement Mono incorpore aussi des composants logiciels pour la surveillance des événements (vue dynamique de comportement).
L'option `--trace` montre au fur et à mesure les noms des méthodes invoquées.

```
...
ENTER: 00 System.String:.cctor (object,intptr,intptr)((nil), (nil), 0xbfffe934, )
. ENTER: 00 System.String:.cctor ()()
.. LEAVE: 00 System.String:.cctor ()
. LEAVE: 00 System.String:.cctor (object,intptr,intptr)[OBJECT:(nil)]
ENTER: 00 System.AppDomain:DoAssemblyLoad (object,intptr,intptr)([System.AppDomain:0x8069fc0], 0xbff-
fec80, (nil), )
. ENTER: 00 System.AppDomain:DoAssemblyLoad (System.Reflection.Assembly)(this:0x8069fc0[System.AppDomain],
[System.Reflection.Assembly:0x80c0ff0], )
...
```

L'outil d'analyse dynamique de performance réalisé utilise une variante plus efficace de ce mécanisme de traçage. Un numéro de méthode et le temps exact sont mémorisés dans un grand tampon de mémoire à chaque entrée ou sortie de méthode (l'annexe I.1). Ceci permet l'analyse détaillée des événements pertinents pour les appels *.Net Remoting*. L'analyse de performance implique une mesure précise de tous les temps d'exécution. Le surcoût introduit par ces mesures ne dépasse pas 16.5% (le tableau 4.1).

Tableau 4.1 Le surcoût d'instrumentation

Nombre d'appels	*Client*	*Client avec trace*	*Surcoût*
1 appel	1.599s	1.859s	16.2%
10 appels	2.215s	2.410s	8.8%
100 appels	9.402s	9.631s	2.4%
1000 appels	1m21.649s	1m22.911s	1.5%

L'exécution d'une méthode appelée par un client plusieurs fois est un processus cyclique: *appel*, *exécution* et *retour*. On retrouve le même caractère cyclique pour les clients distants (*appels en réseau*). Cependant n'oublions pas que dans le cas des applications réparties, la possibilité d'interruption de communication matérielle et logicielle entre les composants peut toujours apparaître.
Pour l'analyse et l'optimisation des temps de réponse pour les appels en réseau, on doit considérer aussi le temps de latence qui implique un temps de réponse accrû en comparaison à un appel local.

Dans l'architecture CLI, un appel *.Net Remoting* (local ou distant) signifie en fait la communication entre *deux domaines d'application différents* (client et serveur). La première étape, *la collecte* des temps d'entrée et de sortie pour chaque méthode invoquée, est effectuée en utilisant l'option `--trace` du compilateur JIT au moment de l'exécution.

Les données collectées concernant chaque méthode sont stockées dans deux tampons (1MB chacun d'eux). Le registre TSC précis au cycle d'horloge près [47], est utilisé pour les valeurs de temps.

```
struct methodInOut {
        char sens;
        unsigned int token;
        unsigned int tscL;
        unsigned int tscH;
};
...
static methodInOut enter[128000], leave[128000];
```

Lorsque plein, chacun des tampons (qui contiennent les informations décrites par la structure définie ci-dessus, correspondant au moment d'entrée ou de sortie de

Tableau 4.2 Nombre de méthodes pour appel local et en réseau

	Appel 1	*Appels 2 - 9*
Local		
Client	1056	776
Serveur	9054	933
Réseau		
Client	1062	774
Serveur	9052	931

chaque méthode) est réinitialisé, est sauvegardé dans un fichier de type binaire, et est réinitialisé pour être réutilisé.

Une fois finie l'interaction client-serveur, commence *l'analyse post-mortem.* L'application utilisée dans les premiers tests est une simple *addition en réparti.* Le client demande à un objet serveur d'additionner deux numéros, n fois (n est le troisième arguments et sera référé comme *nombre des appels*). Les tests ont été effectués pour un nombre différents d'appels (n=1, 10, 100) pour deux situations: les applications client et serveur sont sur le même ordinateur, ou sont en réseau.

Pour un seul appel *.Net Remoting* local, on a identifié 1056 méthodes appelées, dont 972 distinctes du côté client et 9054 méthodes appelées, dont 1029 distinctes du côté serveur.

Pour un appel *.Net Remoting* en réseau, on a identifié un nombre à peu près égal des méthodes invoquées. Une preuve de la flexibilité de l'infrastructure *.Net Remoting* est que le comportement de l'appel local est semblable à celui en réseau (le même nombre de méthodes invoquées). Cela veut dire que la transparence est assurée par l'intergiciel.

Pour dix appels *.Net Remoting*, les résultats sont synthétisés dans le tableau 4.2. Le nombre de méthodes invoquées par le premier appel est visiblement plus grand que pour les appels suivants. Les composants de l'infrastructure *.Net Remoting*, une fois instanciés au premier appel, demeurent dans l'espace d'adressage de chaque

processus (client et serveur). Les informations concernant toutes les méthodes invoquées pendant un appel sont enregistrées dans les fichiers de trace.

Pour gérer la complexité d'une application répartie, entraînée par les nombreux sous-composants (fig. 4.1) qui interagissent, d'une manière prédéfinie, en vertu des patrons englobés dans l'infrastructure, on utilise des méthodes d'abstraction appropriées, telles que le remplacement des événements ou des groupes d'événements primaires dans des *événements abstraits*. Un *événement* décrit l'occurrence de *quelque chose*, qui concerne un objet, dans le système observé à un certain moment.

Une vue statique de la complexité des interactions entre les sous-composants pour notre application de test, *client.exe*, est donnée par l'outil de Mono, *monograph*, à partir du code CIL(fig. 4.1). La figure 4.1 montre les liens entre les types de cette application, mais surtout montre que le niveau de complexité est trop grand à ce niveau de détail.

Un *système réparti* représente une collection des processus où chaque processus comprend une séquence d'événements, un événement pouvant représenter un sous-programme, une instruction machine. L'envoi d'un message représente un événement, la réception d'un message est un autre événement. Dans un système réparti, on peut parler seulement que d'un ordre d'événements (du point de vue global) [54].

Pour un appel *.Net Remoting* simple entre le client et l'objet serveur, on peut définir *quatre événements abstraits* qui décrivent complètement les phases obligatoires d'un cycle d'appel pour le *client*:

$$\texttt{DébutInvocationClient} \longrightarrow \texttt{ClientEnvoieMessage} \rightsquigarrow \ldots \textit{réseau} \ldots \rightsquigarrow$$
$$\texttt{ClientReçoitRéponse} \longrightarrow \texttt{FinInvocationClient}$$

Le même procédé de description est employé par l'*objet serveur*. Les *quatre événements* qui décrivent les phases d'exécution sont les suivants:

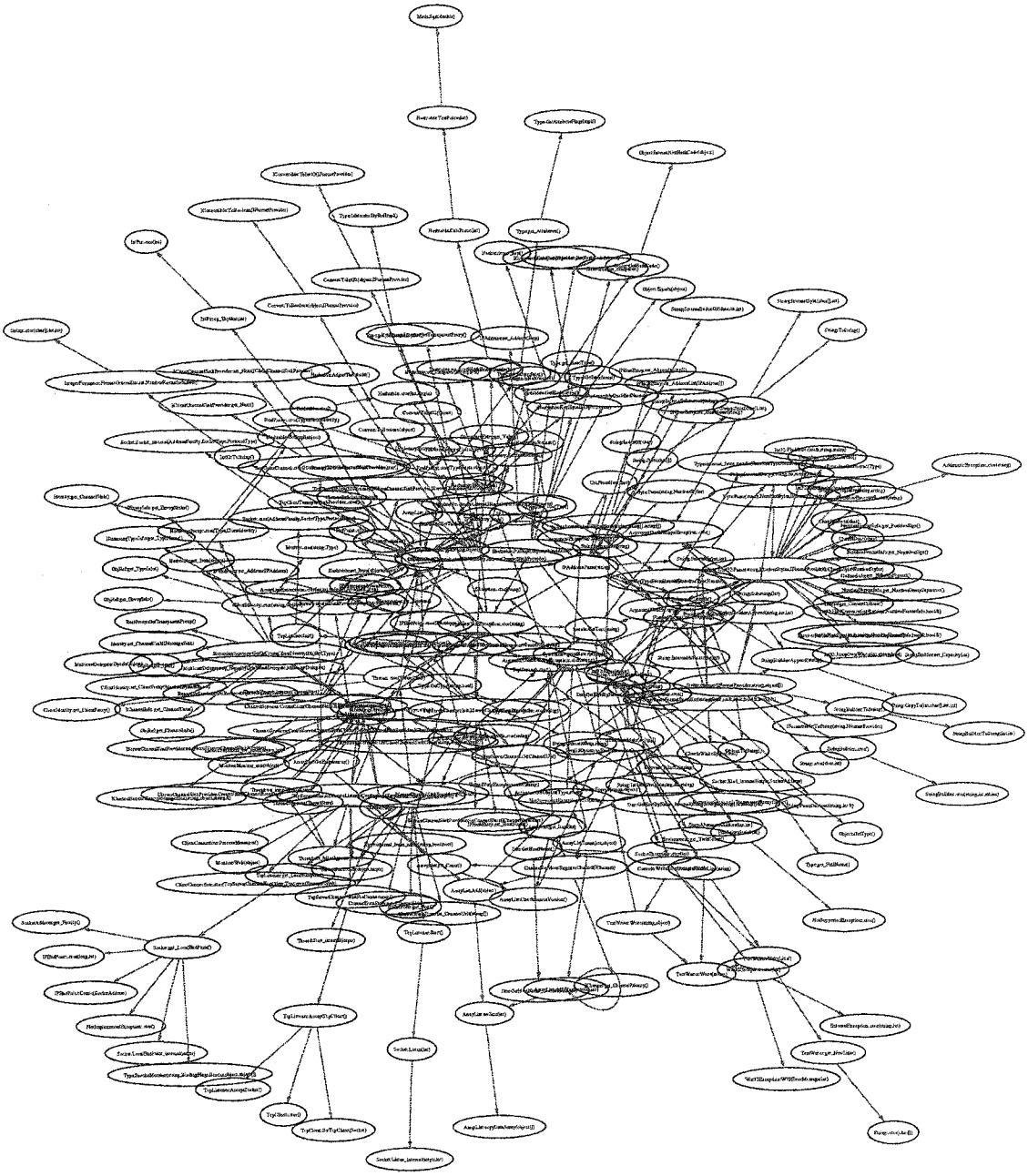

Figure 4.1 Les sous-composants pour l'application client. Le diagramme montre bien le trop grand nombre d'éléments à ce niveau de détail.

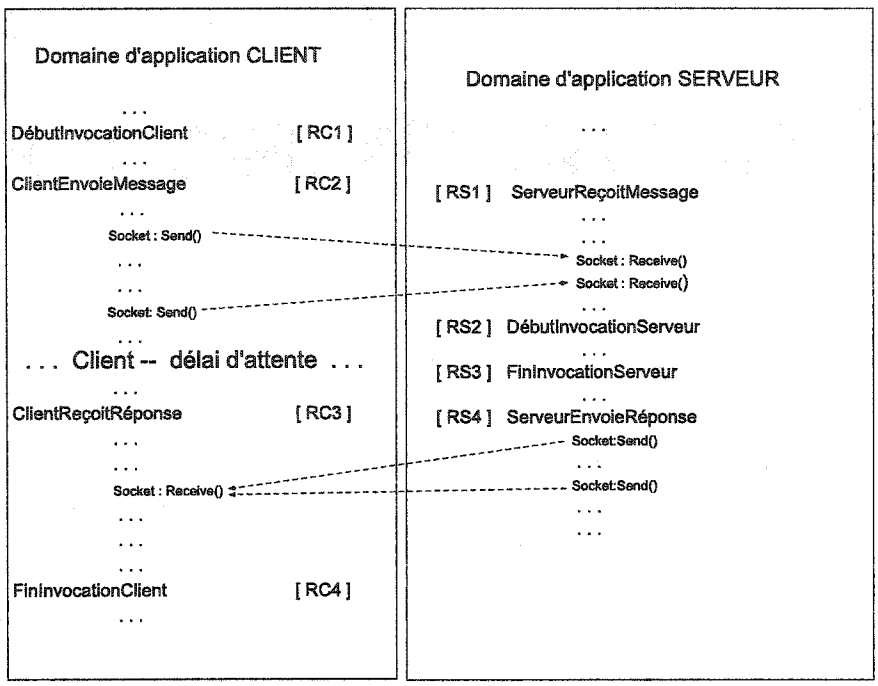

Figure 4.2 Cycle caractéristique d'un appel distant

...*réseau*⇝ `ServeurReçoitMessage` ⟶ `DébutInvocationServeur`
⟶ `FinInvocationServeur` ⟶ `ServeurEnvoieRéponse`⇝ *réseau*...

Une représentation synthétique des événements de chaque côté peut être observée dans les figures 4.2 et 4.3. Le cycle caractéristique pour chaque appel distant est défini par la séquence suivante (et toute séquence complète décrivant un appel *.Net Remoting*):

`RC1` ⟶ `RC2` ⇝ *réseau*⇝ `RS1` ⟶ `RS2` ⟶ `RS3` ⟶ `RS4` ⇝ *réseau*⇝ `RC3` ⟶ `RC4`

Les *événements primitifs* (méthodes) qui ont été remplacés par les *événements abstraits* présentés sont:

Du côté client:

RC1 = RemotingProxy:Invoke

RC2 = TcpMessageIO:SendMessageStream

RC3 = TcpMessageIO:Receive MessageStream

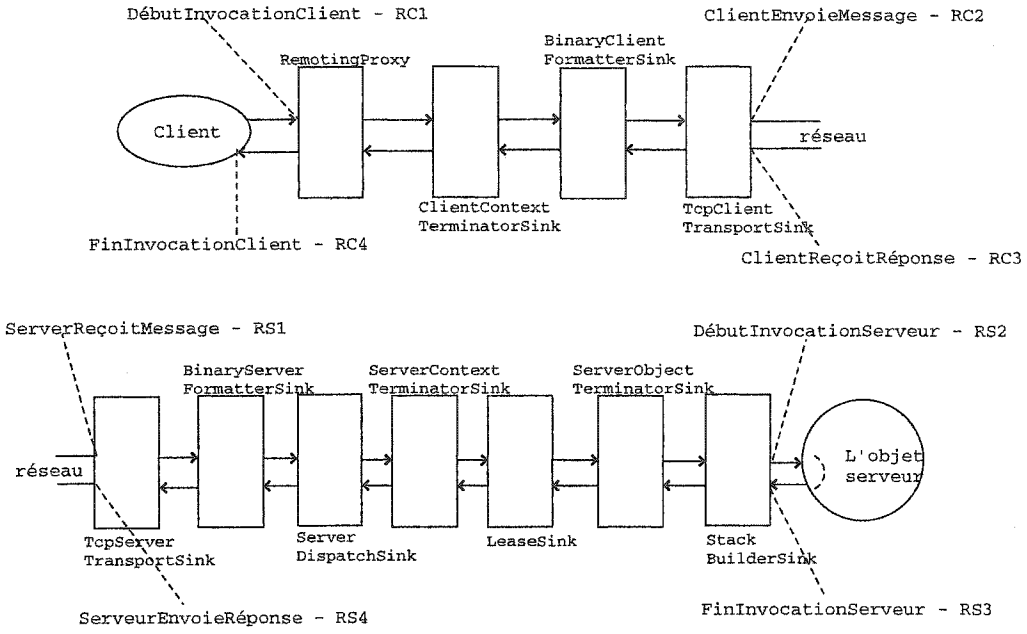

Figure 4.3 La traduction d'événements primaires en événements abstraites

RC4 = ReturnMessage:get_Exception

Du côté serveur:

RS1 = TcpServerTransportSink:InternalProcessMessage

RS2 = RemotingServices:InternalExecuteMessage

RS3 = ReturnMessage:.ctor

RS4 = TcpMessageIO:SendMessageStream

La figure 4.3 montre dans quels points de l'infrastructure Remoting se produisent ces événements abstraits.

On connaît ainsi les moments d'entrée en scène de tous les sous-composants cachés de l'infrastructure *.Net Remoting* et, s'il y a lieu, les moments de sortie. À partir de ces huit repères, sont faits l'analyse et le traitement des fichiers de trace, à l'aide du programme conçu. Les exceptions et autres sorties de fonctions manquantes (*tail call elimination*) sont aussi tenues en compte lorsqu'elles surgissent

du côté client ou serveur.

Le fichier final contient les champs d'informations suivants: un indicateur unique pour chaque méthode (token), la profondeur (depth) qui désigne le niveau d'imbrication par rapport au parent, le temps total, le temps accumulé par les enfants, le temps propre (self), le nombre d'enfants, le moment d'entrée et le nom de la méthode.

```
#CALL-1 has completeCall = [True] and [ 3 ] exceptions
#137046368 / TcpConnectionPool:ConnectionCollector
#136455104 / MulticastDelegate:invoke_void
#135292256 / Thread:Sleep
# 0—- [123.967] — Server — [706.923]—- [736.738]
136506704 0 40 736.540 730.075 6.465 13 6.465 RemotingProxy:Invoke 165916743.647
136725200 1 39 0.012 0.000 0.012 0 6.476 MonoMethodMessage:get_MethodBase 165916743.873
136387688 1 39 0.385 0.023 0.362 1 6.838 MethodBase:get_IsConstructor 165916744.347
136392320 2 38 0.023 0.000 0.023 0 6.862 MonoMethod:get_Attributes 165916744.708
...

#CALL-2 has completeCall = [True] and [ 0 ] exceptions
# 0—- [2.766] — Server — [14.158]—- [15.880]
136506704 0 40 15.879 15.865 0.014 13 1359.616 RemotingProxy:Invoke 165917480.896
136725200 1 39 0.003 0.000 0.003 0 1359.619 MonoMethodMessage:get_MethodBase 165917480.901
136387688 1 39 0.016 0.011 0.005 1 1359.624 MethodBase:get_IsConstructor 165917480.905
...
#CALL-3 has completeCall = [True] and [ 0 ] exceptions
# 0—- [2.406] — Server — [42.264]—- [76.610]
136506704 0 40 76.608 76.596 0.012 13 1375.493 RemotingProxy:Invoke 165917496.885
136725200 1 39 0.003 0.000 0.003 0 1375.496 MonoMethodMessage:get_MethodBase 165917496.889
136387688 1 39 0.016 0.011 0.004 1 1375.501 MethodBase:get_IsConstructor 165917496.893
...
```

Les deux programmes de traitement (un pour les clients et un pour les serveurs) sont développés en C# (les annexes II.1, II.2). Pour les tests effectués sur la même machine (deux domaines d'application), la visualisation inclura le client et le serveur sur le même diagramme. Pour les tests en réseau, la synchronisation des horloges entre les deux systèmes s'impose avant la visualisation.

Les tests ont été effectués dans le laboratoire CASI sur les ordinateurs *Mario* et *Wolfsburg* dans un environnement isolé du réseau polymtl.ca, afin de minimiser le temps de latence.

Les deux registres TSC ont des valeurs différentes. Il faut les convertir à une base de temps commune. Le schéma de la figure 4.4 représente d'une manière simplifiée

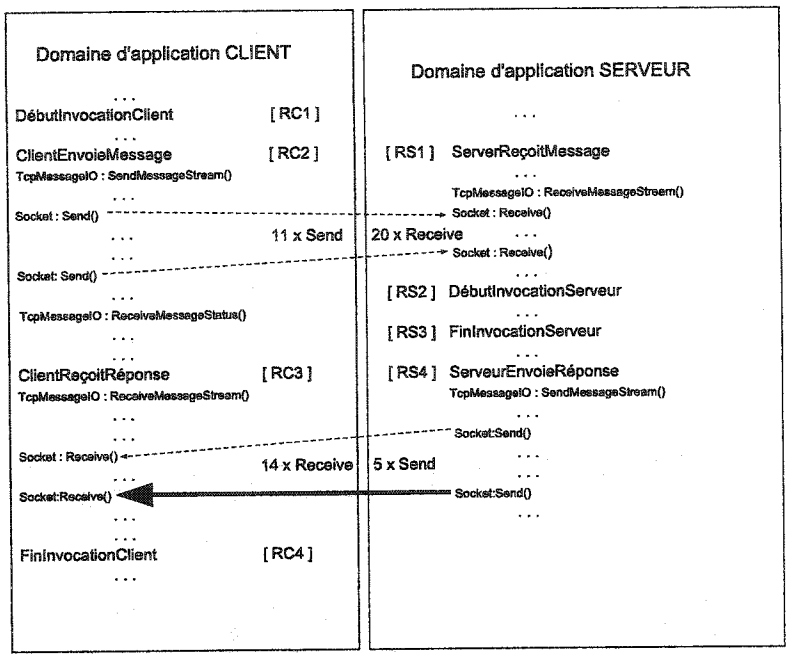

Figure 4.4 L'interaction client - serveur

l'interaction entre les objets de l'application répartie. Pour chaque cycle d'appel distant, on suppose que le dernier appel *Send()* du serveur et le dernier appel *Receive()* du côté client (fig. 4.4) se produisent simultanément [54], le réseau local étant très rapide. Ainsi, on peut calculer les valeurs de temps pour le client et pour le serveur et les représenter dans le même diagramme.

Suit la dernière étape de l'analyse de performances: la visualisation. Elle donne une image plus précise de chaque composant, condition essentielle pour la compréhension de comportement de systèmes répartis. Pour la représentation graphique on a choisi *gnuplot* existant sur toutes les distributions Linux. On a découpé les événements et on a identifié les composants internes et les méthodes spécifiques en utilisant des filtres (l'annexe III.1). L'outil d'analyse de performance présenté est une variante simplifiée, et bien sûr perfectible, d'un outil d'analyse idéal. Il représente un compromis entre la simplicité de la méthode utilisée, la minimisation des modifications nécessaires pour l'environnement, et la précision des résultats.

On peut remarquer que l'outil d'analyse, sous cette forme, pourrait couvrir des aspects qui n'ont pas été abordés comme:

- l'analyse de performances de l'exécution des fils génériques et de ceux appartenants au ThreadPool;
- l'analyse de la performance de la communication utilisant HttpChannel,
- la performance des canaux différents comme: RMI, IIOP [4].

Une fois les données collectées et stockées, suivra la phase de visualisation et d'interprétation, qui donne une image plus précise des événements produits dans le système.

CHAPITRE 5

LES RÉSULTATS

Les tests ont été effectués avec le client et l'objet serveur sur la même machine et en réseau respectivement, pour les appels synchrones. Les caractéristiques des machines de test sont:

- Mario: PIII - 450MHz, 256 MB, Slackware 9.0, Mono 0.25 et
- Wolfsburg: PIII - 550MHz, 256 MB, RedHat 9.0, Mono 0.25.

Le cycle d'exécution d'un appel *Remoting* est étudié à l'aide des huit repères:

RC1 $\rightarrow$ RC2 $\rightsquigarrow$ *réseau* $\rightsquigarrow$ RS1 $\rightarrow$ RS2 $\rightarrow$ RS3 $\rightarrow$ RS4 $\rightsquigarrow$ *réseau* $\rightsquigarrow$ RC3 $\rightarrow$ RC4

Le cas des appels locaux est presenté en premier. Ci-dessous sont montrées les séquences d'événements pour le premier appel (fig. 5.1), le deuxième (fig. 5.2) et le troisième appel (fig. 5.3).

Une première observation, le temps de réponse du côté client (RC4 – RC1) est de respectivement 736.738, 15.880 et 76.610 ms, visiblement plus grand que le temps nécessaire à une information pour qu'elle fasse un aller-retour entre le client et le serveur (environ 0.1 ms).
Pour les trois premiers appels, les temps du côté serveur (RS4 – RS1) sont: 561.862, 9.948 et 38.463 ms.

Le premier appel est crucial dans la communication, c'est pourquoi il est le plus grand consommateur de temps. Pendant cet appel, sont instanciés de chaque côté tous les sous-composants spécifiques de l'architecture *.Net Remoting* (figures 4.1 et 4.3) dont les fournisseurs de récepteurs et la chaîne des fournisseurs qui vont instancier les récepteurs. Chacun d'eux accomplira sa tâche, soit permettre l'échange de messages entre les deux participants au dialogue.

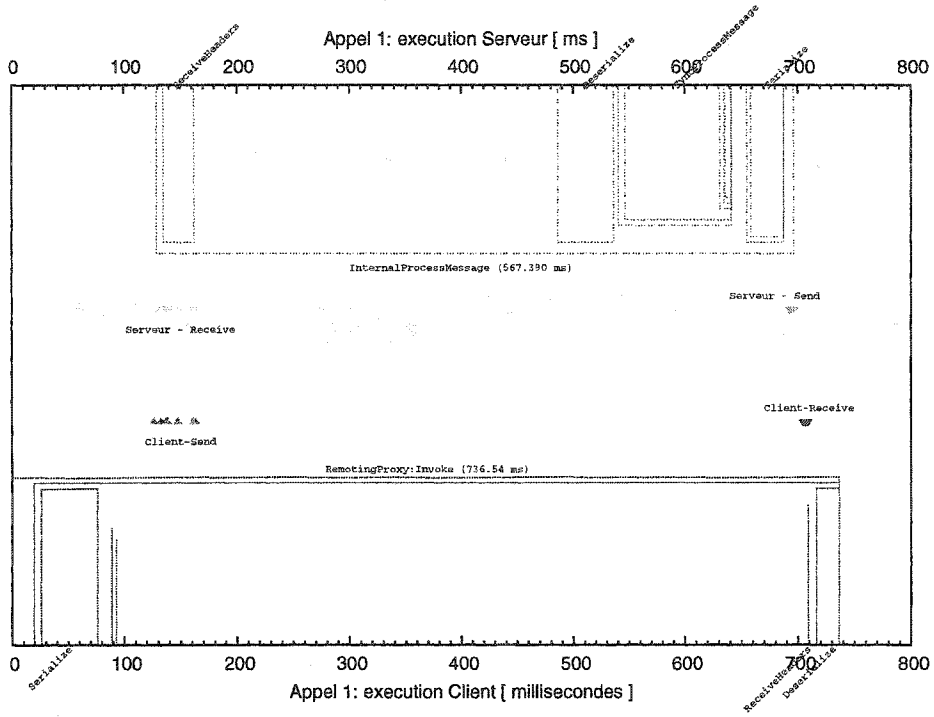

Figure 5.1 Appel 1 en local

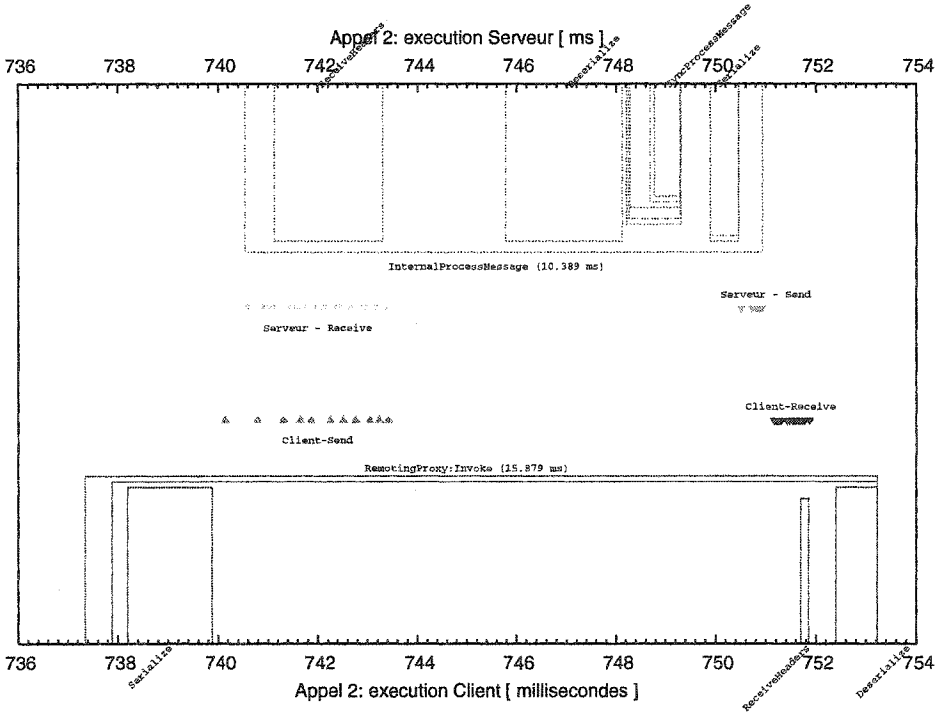

Figure 5.2 Appel 2 en local

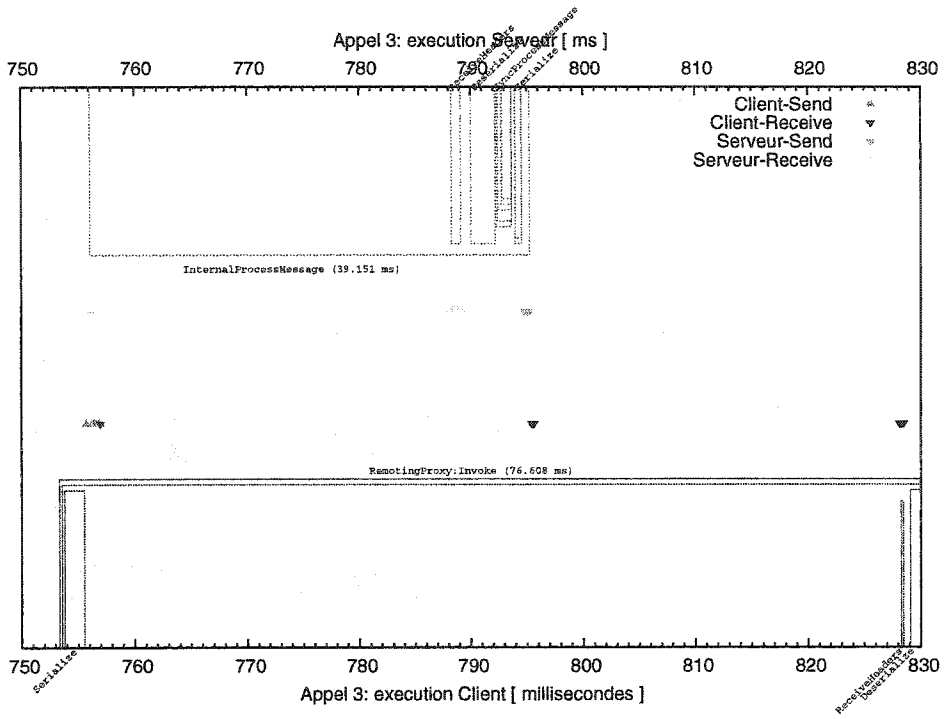

Figure 5.3 Appel 3 en local

Pour le premier appel, on retrouve de côté client 1059 méthodes invoquées, dont 122 sont des constructeurs, comparativement au deuxième et troisième appel: 774 méthodes invoquées, dont 53 constructeurs. Du côté serveur sont invoquées pour le premier appel 9065 méthodes, dont 410 constructeurs, pendant que pour le deuxième et troisième appel sont invoquées 932 méthodes, dont 65 constructeurs.

Le temps passé du côté client est dû aux processus comme: la sérialisation qui utilise la réflexivité pour trouver les noms, les types et les paramètres; l'encapsulation de ces informations dans un message; l'envoi de message (tous les trois compris dans l'intervalle définit par: (RC2 – RC1)); l'attente (RC3-RC2) et la désérialisation de la réponse (RC4-RC3).

Les valeurs de temps obtenues pour la sérialisation (incluant les enfants appelés) sont:

- du côté client: appel 1) 50.318 ms, appel 2) 1.685 ms, appel 3) 1.756 ms,
- du côté serveur: appel 1) 32.496 ms, appel 2) 0.562 ms, appel 3) 0.559 ms.

Le formateur ajoute des informations d'identification pour l'assemblage (version, culture) et le nom complet de *Type*. Celles-ci sont réunies et envoyées comme un flot de données. À la désérialisation, les opérations inverses ont lieu.
Les temps obtenus pour la désérialisation sont:

- du côté client: 20.478, 0.820 et 0.901 (ms)

- du côté serveur: 50.272, 2.346 et 2.173 (ms)

L'ouverture d'une connexion sur l'objet serveur par le client, effectuée pendant le premier appel, est aussi un grande consommatrice de temps: *TcpConnectionPool:GetConnection()* 36.311 ms comparativement avec les appels ultérieurs: 0.169 ms et 0.163 ms.

Une fois le message envoyé par le client, du *côté serveur* ont lieu la création de la chaîne de récepteurs, l'écoute des requêtes, la désérialisation des requêtes, la création d'objets ou la recherche d'objets serveur, l'exploration par réflexivité pour la méthode appelée (pendant l'intervalle RS2–RS1); l'invocation de la méthode réelle (RS3–RS2), la sérialisation et l'envoi de la réponse (RS4–RS3).

Du côté serveur, il y a la même distribution de participation des méthodes consommatrices de temps appartenant à l'assemblage *System.Reflection.*
On rencontre pour le premier appel:

- 1800 invocations pour *GetType()* et seulement 8 invocations pour les appels suivants;

- 1800 invocations pour *CompareTo()* et aucune pour les appels suivants.

L'absence d'invocation de ces méthodes pour les appels 2...10 est due au compilateur JIT et à la structure de métadonnées de chaque *Type*. Après la première invocation, ces méthodes seront seulement référées en accédant au code natif produit lors du premier appel (fig. 3.6 et fig. 3.4).

Après la désérialisation, la méthode *SyncProcesMessage()* est invoquée sur toute la chaîne existante jusqu'à l'objet serveur: *CrossContextChannelSink, ServerContextTerminatorSink, LeaseSink, ServerObjectTerminatorSink, StackBuilderSink*. À ce moment se produit l'appel de la méthode invoquée par le client. La durée de cette méthode, pour le premier appel, est de 94 ms, et pour les appels suivants 1.041 ms et 1.241 ms. Le résultat d'invocation de *SyncProcesMessage()* est un objet de type *ReturnMessage* contenant le résultat d'invocation de l'appel distant, qui est sérialisé et envoyé au client, fermant ainsi le cycle d'appel.
Les causes des différences entre les appels ont été étudiées.

Les appels 2, 3...10 sont similaires du point de vue du nombre des méthodes invoquées. Cependant le deuxième appel présente certaines particularités. Il semble le plus court (15 ms de côté client). Cette durée est exprimée comme le temps écoulé entre les moments de début de RC1 et la fin de RC4, le cycle qui comprend la participation du serveur.

Nous analyserons l'apport du client à cet échange de messages pour les appels en réseau, étant donné que les appels locaux ont quand même un comportement atypique. Les messages sautent le *TcpChannel* et passent directement dans le récepteur *ServerContextTerminatorSink* de l'assemblage serveur.
L'appel 2 présente le plus petit impact de type E/S. Si on fait le rapport entre le temps passé dans la méthode *Socket.Send()*, ou *Socket.Receive()* selon le cas, sur le temps total d'un appel Remoting, on obtient le résultat suivant:
Appel 1: client 73% et serveur 0.01%
Appel 2: client 48% et serveur 0.02%
Appel 3: client 50% et serveur 83%
Appel 4: client 53% et serveur 82%
Appel 5: client 60% et serveur 51%
...

La synchronisation est la meilleure dans le cas de l'appel 2. Celui-ci est le moins tributaire des entrées-sorties. Un comportement similaire a été identifié pour les tests en réseau. Il serait utile de faire une telle analyse pour les appels asynchrones (associés, par défaut, avec de meilleures performances). Cependant, le support pour les appels asynchrones est disponible seulement dans la version .Net, Mono étant en cours de finalisation pour cette option.

Des techniques d'optimisation de la transmission de données pourraient minimiser les temps d'attente du côté client, en évitant les copies supplémentaires des tampons entre le noyau et l'espace usager.

Dans le cas *des appels en réseau* (les figures 5.4, 5.5) le comportement est similaire (le même nombre de méthodes sont invoquées de chaque côté), les temps obtenus sont:

- du côté client, les premiers trois appels: 596.131, 45.435 et 79.896 ms
- du côté serveur: 489.536, 8.228 et 45.312 ms.

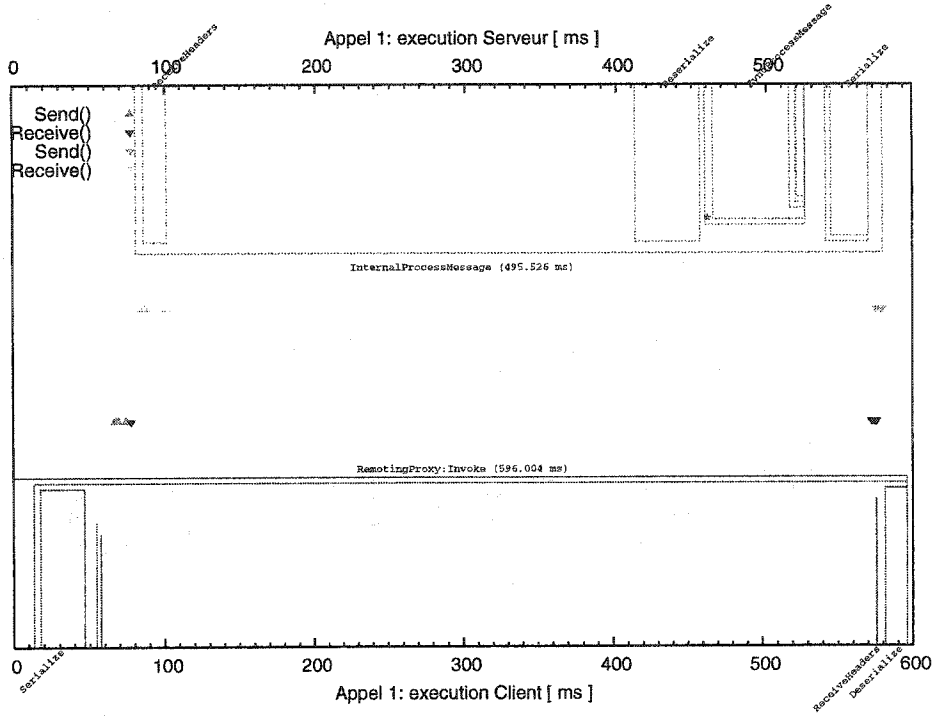

Figure 5.4 Appel 1 en réseau

Toutes les observations faites pour les tests locaux se retrouvent pour les appels entre deux hôtes différents: la structure et la longueur d'appel 1 est particulière, l'appel 2, même avec une structure semblable à celle des appels suivants, frappe par son temps très court, les appels 3...10 sont de durée comparable.

Si on regarde du point de vue de la participation exclusive du côté client à cet échange de messages, cela veut dire que pour les intervalles définis par (RC2–RC1) + (RC4–RC3), on observe que le temps pendant lequel le client est actif décroît à chaque appel et se stabilise après le troisième appel.

Appel 1: 86.724

Appel 2: 36.201

Appel 3: 33.816

Appel 4: 34.213

Appel 5: 34.089

Appel 6: 33.981

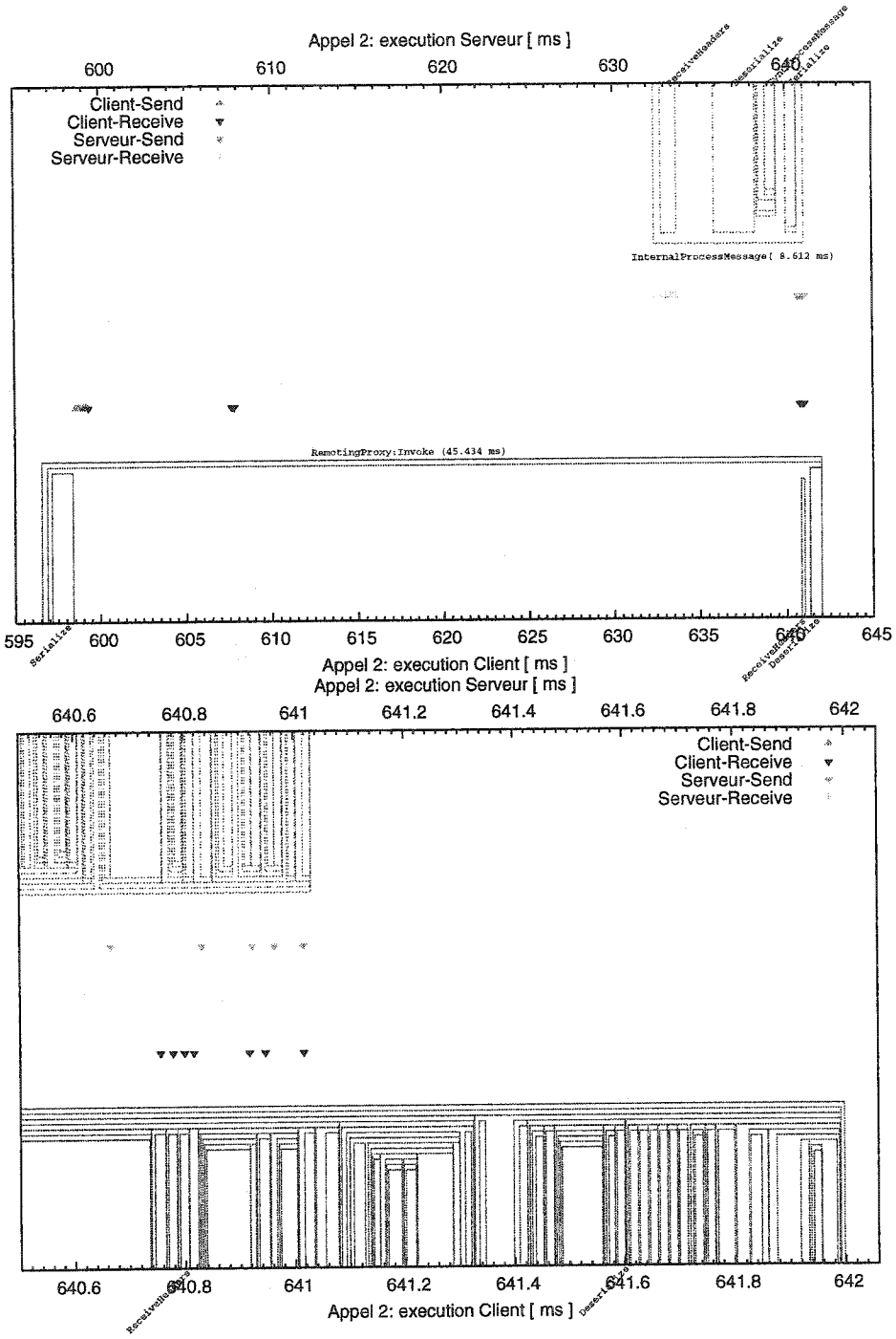

Figure 5.5 Appel 2 en réseau et les détails de fin du dialogue

Appel 7: 33.967

Appel 8: 33.995

Appel 9: 34.266

Appel 10: 34.092

En ce qui concerne l'impact des entrées – sorties, les valeurs obtenues sont:

Appel 1: client 83% et serveur 0.0%

Appel 2: client 72% et serveur 0.01%

Appel 3: client 56% et serveur 85%

Appel 4: client 56% et serveur 86%

Appel 5: client 56% et serveur 86%

...

Toutes les traces visualisées représentent l'activité en mode usager. Des indices concernant certaines invocations de méthodes ayant des temps d'exécution trop grands pourraient être obtenus avec des traces du noyau par exemple avec LTT [69].

Les résultats obtenus et interprétés tout au long de ce chapitre sont synthétisés à la figure 5.6.

L'outil présenté offre une réponse exacte pour une question sous-entendue dans le cas d'une application répartie: le temps passé du côté client et du côté serveur, en étudiant la fonctionnalité de chaque composant. C'est une analyse qui relève les temps détaillés pour l'exécution, par un ensemble de moniteurs efficaces ciblant des aspects bien définis, tels que la sérialisation/déserialisation, l'envoi/reception des messages et le temps d'attente du client.

Un exemple d'optimisation des composants d'une application répartie, en analysant l'impact sur le temps de réponse et le temps de traitement d'une requête est illustré par les tests suivants: un objet serveur traite une requête de tri pour un tableau de grande dimension.

L'objet serveur implémente les méthodes suivantes:

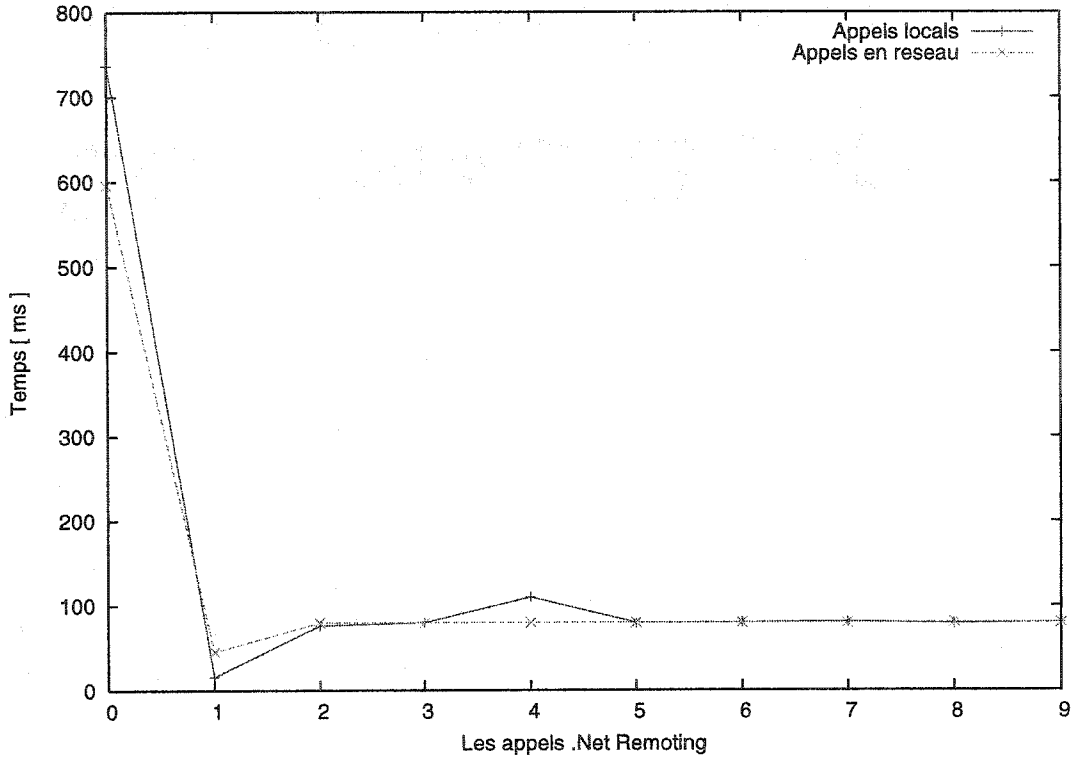

Figure 5.6 Le temps de réponse - en local et en réseau

- Test 1, *int [ ] MethodBubbleSort (LinkedList l);* méthode de tri par permutation d'une liste chaînée de 10000 éléments ;
- Test 2, *int [ ] MethodQuickSort (LinkedList l);* méthode de tri par segmentation d'une liste chaînée de 10000 éléments;
- Test 3, *int [ ] MethodQuickSort (int [ ]);* méthode de tri par segmentation d'un vecteur de 10000 entiers.

Pour tous les trois cas, les tests ont été effectués sur le même ordinateur et en réseau, et à chaque fois le client fait deux appels *.Net Remoting.*

Comme à l'habitude, les étapes pour chaque appel sont: la conversion des objets en octets (sérialisation) et la conversion des octets en objets (désérialisation) du côté client ainsi que du côté objet serveur, et l'exécution réelle de la méthode.

Pour le premier cas de test, l'analyse effectuée relève des valeurs très élevées pour les temps écoulés correspondant à chacune de ces étapes. Ceci est dû au nombre élevé d'éléments transférés, ainsi qu'au manque d'optimisation des composants.

Pour les deux cas de tests suivants, l'application a été optimisée suite aux résultats précédants.

Pour le deuxième cas de test, la méthode de tri par permutation est changée pour une par segmentation. L'analyse indique que les temps de sérialisation / désérialisation de chaque côté sont semblables à ceux du premier cas de test. Cependant, le temps d'exécution de la méthode est de 35 à 40 fois plus petit que dans le cas du tri par permutation, et ceci pour les deux appels en réseau ou sur le même ordinateur.

La représentation graphique montre la différence entre le temps écoulé dans la sérialisation du côté client, la désérialisation du côté serveur pour 10000 éléments dans une liste chaînée, la sérialisation du côté serveur, la désérialisation du côté client, pour le même nombre d'éléments triés.

Le troisième cas de tests a pour but d'analyser l'impact de la sérialisation d'un objet. En remplaçant les 10000 éléments de la liste chaînée par un vecteur d'entiers, l'analyse montre que les temps de sérialisation du côté client, désérialisation du côté serveur sont de 15 à 30 fois plus petits.

Tous les résultats obtenus sont condensés dans les tableaux 5.1 et 5.2.

Tableau 5.1 L'appel numéro 1

	Client-Serialize	*Serveur*-Deserialize	*Serveur*-SyncProcessMessage	*Serveur*-Serialize	*Client*-Deserialize
Local					
Test 1	8470.102	12210.348	2785.837	596.307	417.630
Test 2	8435.862	12405.082	71.204	602.251	443.102
Test 3	563.597	399.606	72.372	527.306	367.981
Réseau					
Test 1	8496.469	23573.481	3507.008	725.795	412.651
Test 2	8395.840	25074.894	99.498	821.976	416.550
Test 3	560.791	601.001	98.205	801.367	368.817

Tableau 5.2 L'appel numéro 2

	Client-Serialize	*Serveur*-Deserialize()	*Serveur*-SyncProcessMessage	*Serveur*-Serialize	*Client*-Deserialize
Local					
Test 1	8427.990	12208.394	2781.326	435.977	386.798
Test 2	8414.547	12181.789	65.717	486.586	423.866
Test 3	483.527	370.770	62.449	495.203	386.389
Réseau					
Test 1	8425.330	30710.423	3501.086	747.547	376.987
Test 2	8400.956	27297.894	95.560	705.321	380.017
Test 3	484.201	528.019	95.984	696.895	379.672

Les figures 5.7, 5.8 et 5.9 représentent le deuxième appel *.Net Remoting* pour les trois cas de tests. Une comparaison entre les deux premiers montre les différences au niveau de la méthode de tri. Une comparaison entre les cas de tests 2 et 3 montre les différences au niveau de la sérialisation et de la désérialisation. La conversion objets–octets et octets–objets est extrêmement coûteuse. L'infrastructure CLI génère un nouvel objet *SerializationInfo* et peuple les champs avec les éléments de la liste et les associations entre eux. Toutes ces opérations se font par des appels à l'espace de nom *System.Reflection*, qui est le principal consommateur de temps.

Les figures 5.10 et 5.11 montrent les insuffisances du processus de désérialisation/sérialisation du côté serveur, à cause de la réflexivité. Les figures 5.12 et 5.13 montrent l'effet du JIT sur le premier appel en le comparant avec le cas du second appel. Les effets de l'optimisation d'un cas de test par rapport à l'autres peuvent être observés sur la figure 5.14.

Dans tous les tests effectués, on peut constater que les appels multiples, nécessaires pour envoyer un message logique, impliquent plusieurs appels systèmes. Chaque appel système peut génèrer un changement de contexte, et un ré-ordonnancement, ce qui implique des délais significatifs. En pratique, on désire généralement envoyer/recevoir le plus d'informations possible pour réduire le nombre d'appels et de paquets échangés. Présentement, le dialogue de Mono/Remoting commence avec trois paquets TCP, chacun contenant un seul caractère: 'N', 'E' et respectivement 'T'. En ajoutant un tampon (*BufferedStream*) dans l'implantation du canal TCP nous avons découvert que la performance des appels simples pouvait être multipliée par 100. Cette amélioration a été communiquée aux developpeurs de Mono et sera incluse dans une version ultérieure.

Ce problème de performance un peu exceptionnel était difficile à identifier precisément avec le nouvel outil proposé, ou avec un autre outil, puisque son impact se faisait sentir à chaque envoi de paquet, ce qui est réparti à travers plusieurs des phases du cycle d'appel Remoting.

Dans l'environnement Mono/.Net, tous les aspects de bas niveau de la programmation n'ont pas disparu, mais sont plutôt encapsulés. Ils se retrouvent dans le contexte d'exécution CLI. Suite à une analyse, l'optimisation d'une application nécessite parfois la modification de certains composants de l'environnement.

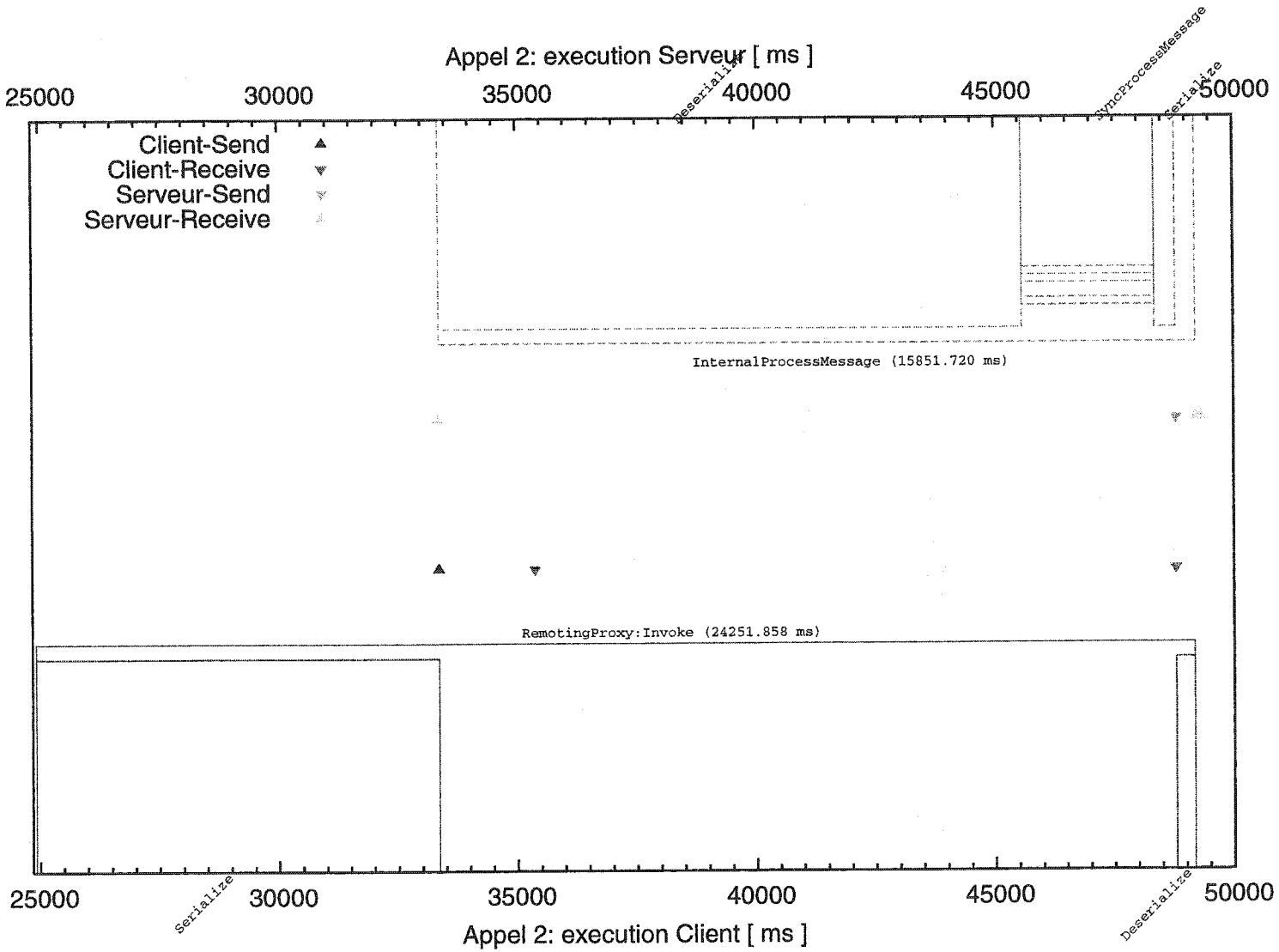

Figure 5.7 Appel 2 en local - Test 1

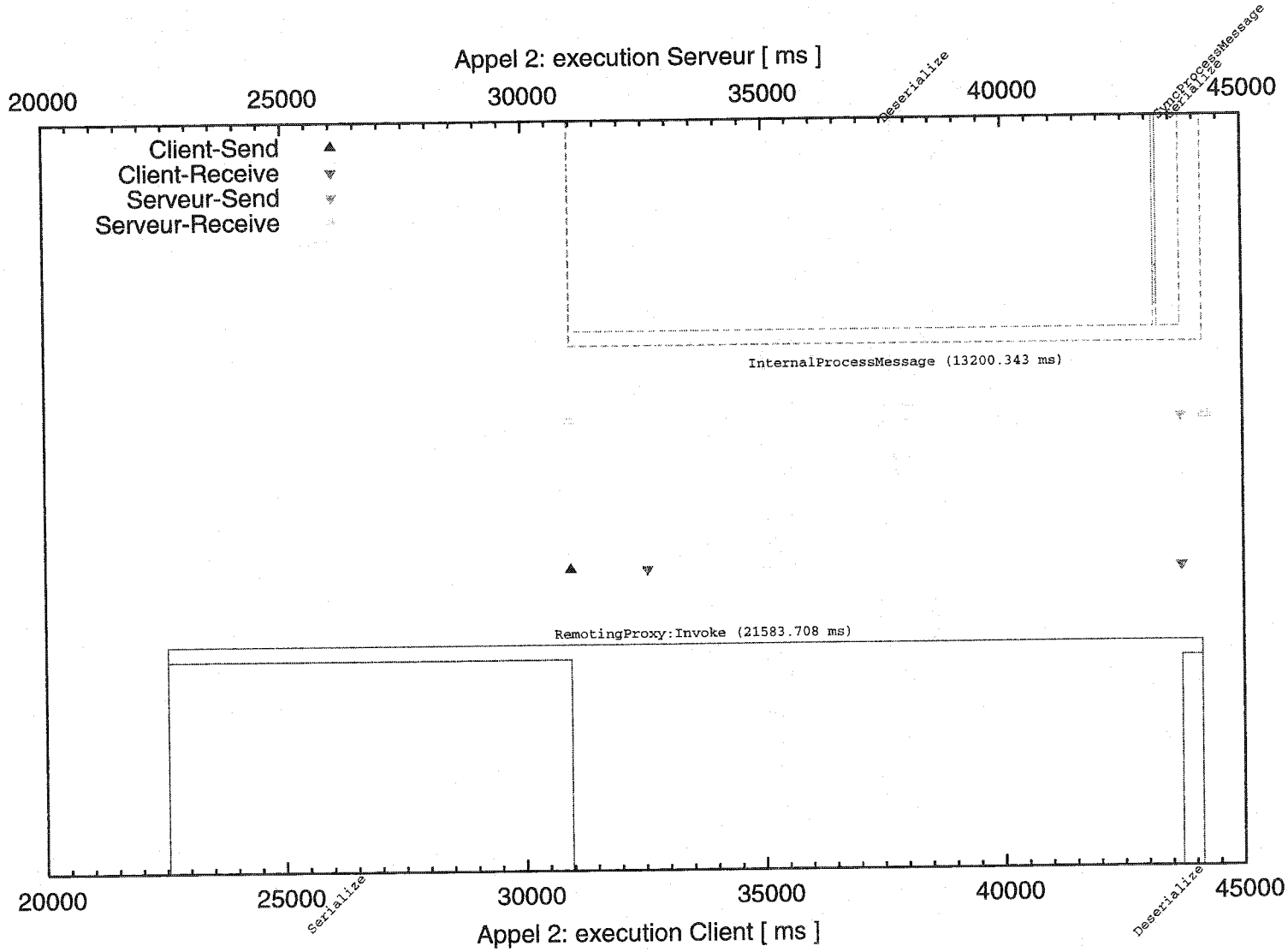

Figure 5.8 Appel 2 en local - Test 2

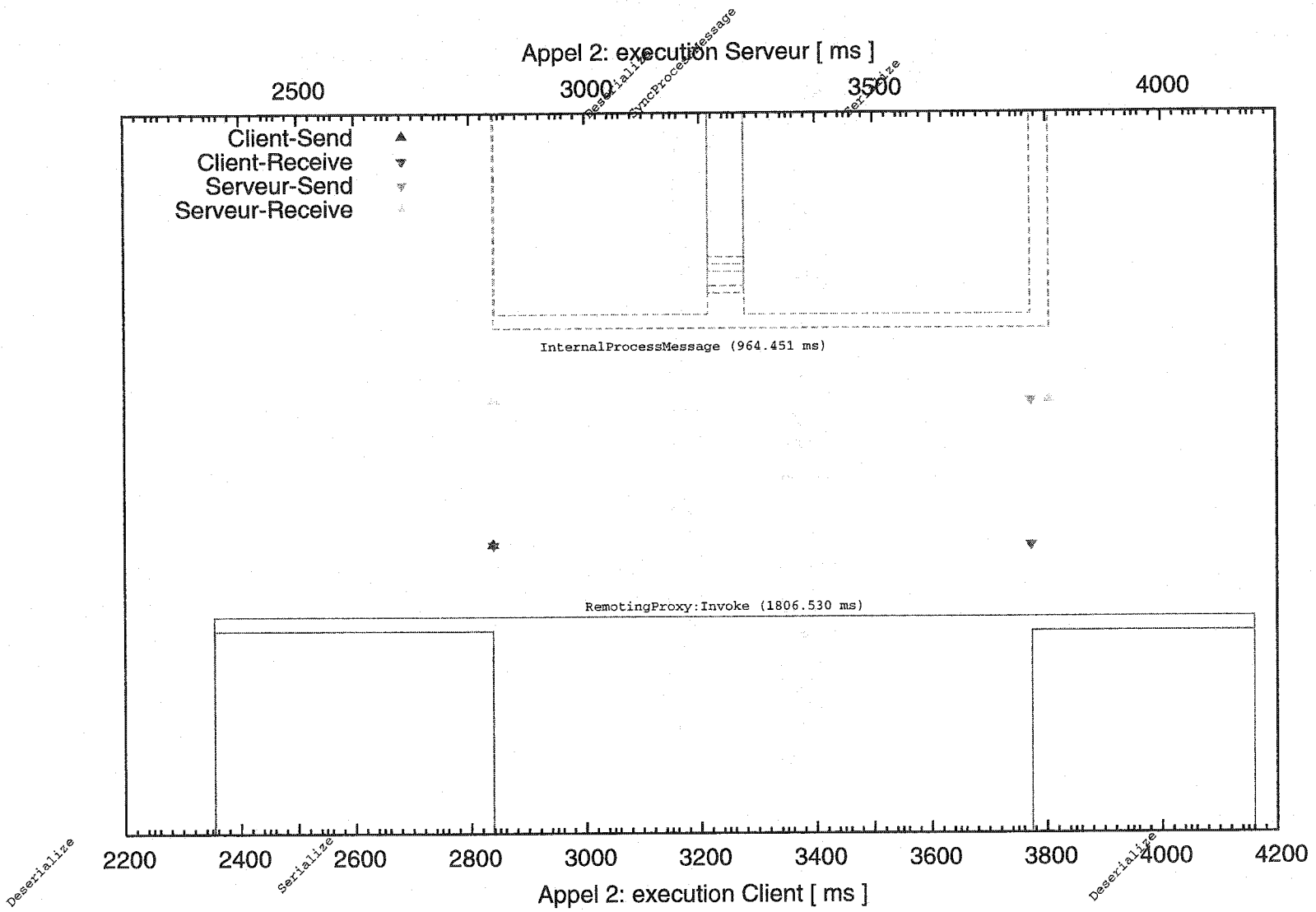

Figure 5.9 Appel 2 en local - Test 3

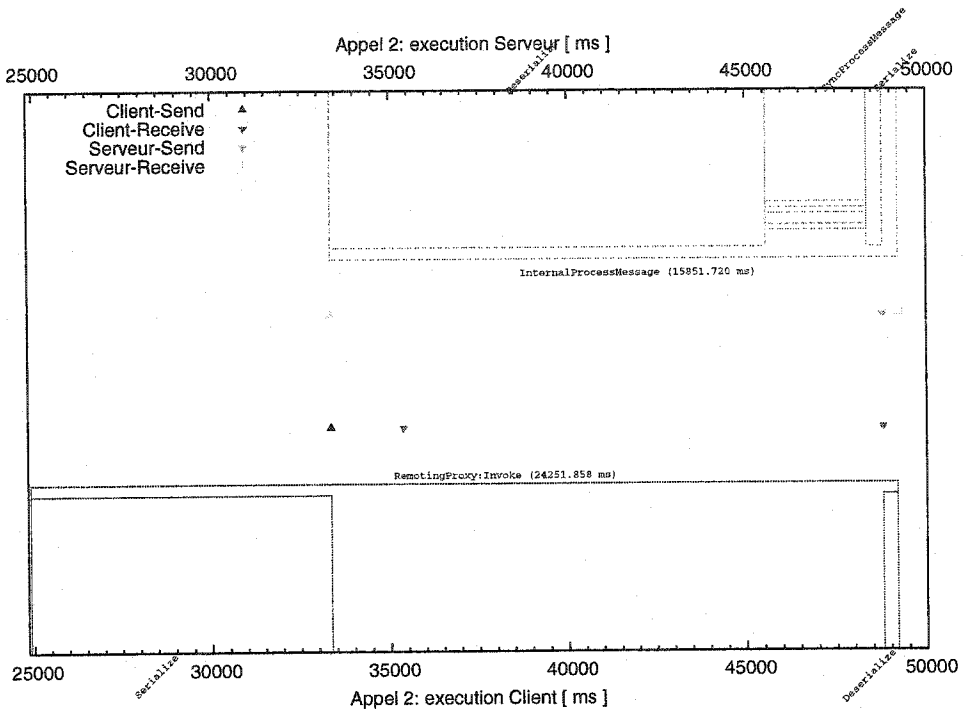

Figure 5.10 Appel 2 en local - Test 1

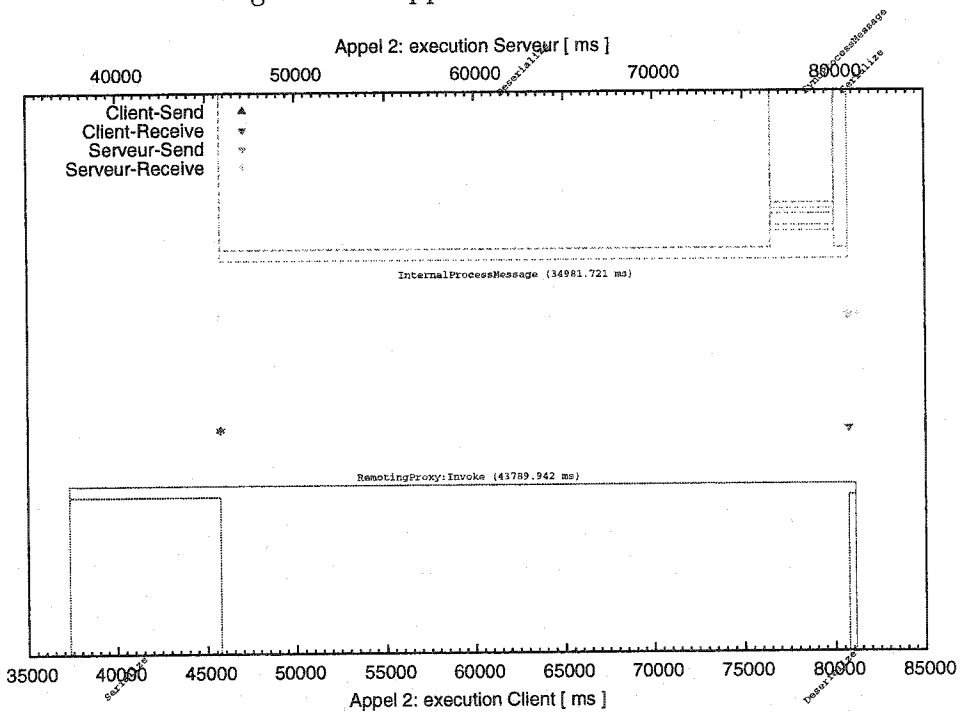

Figure 5.11 Appel 2 en réseau - Test 1

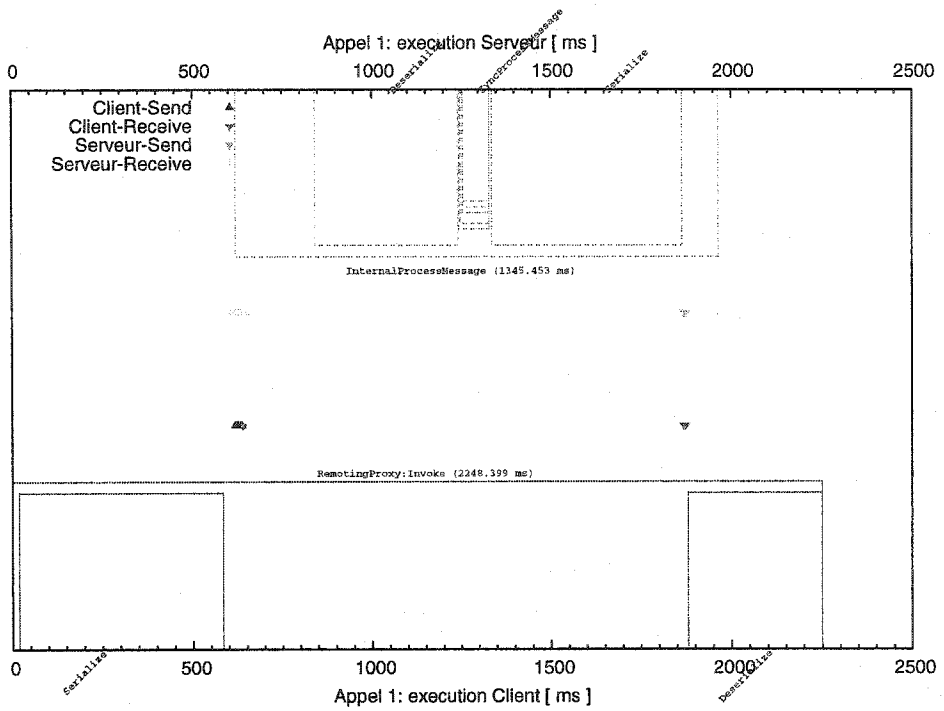

Figure 5.12 Appel 1 en local - Test 3

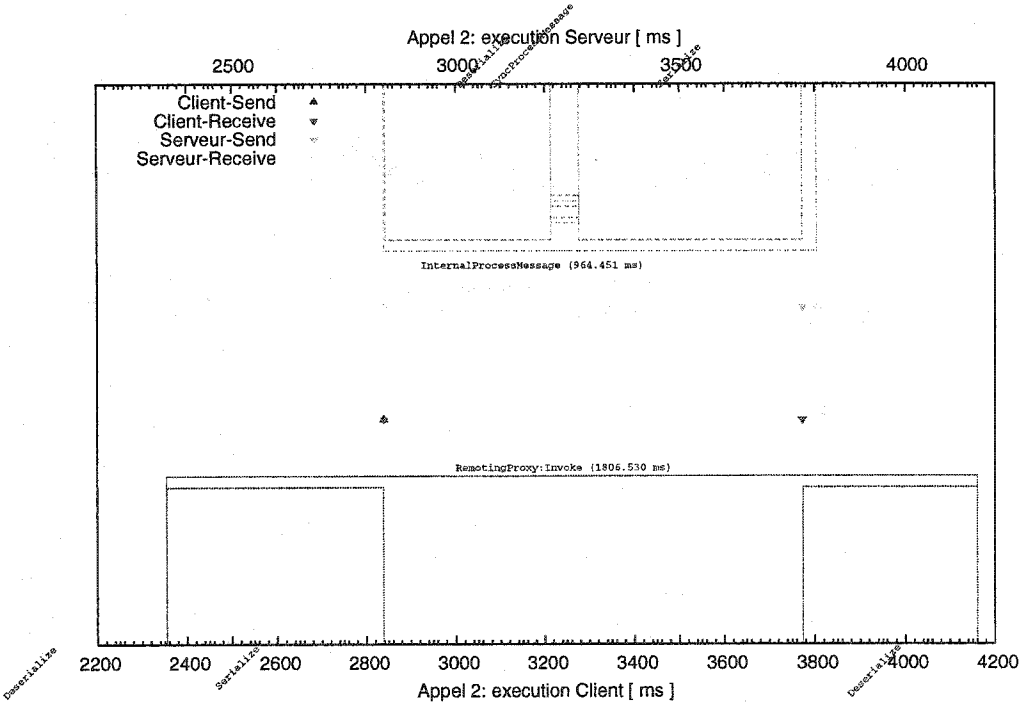

Figure 5.13 Appel 2 en local - Test 3

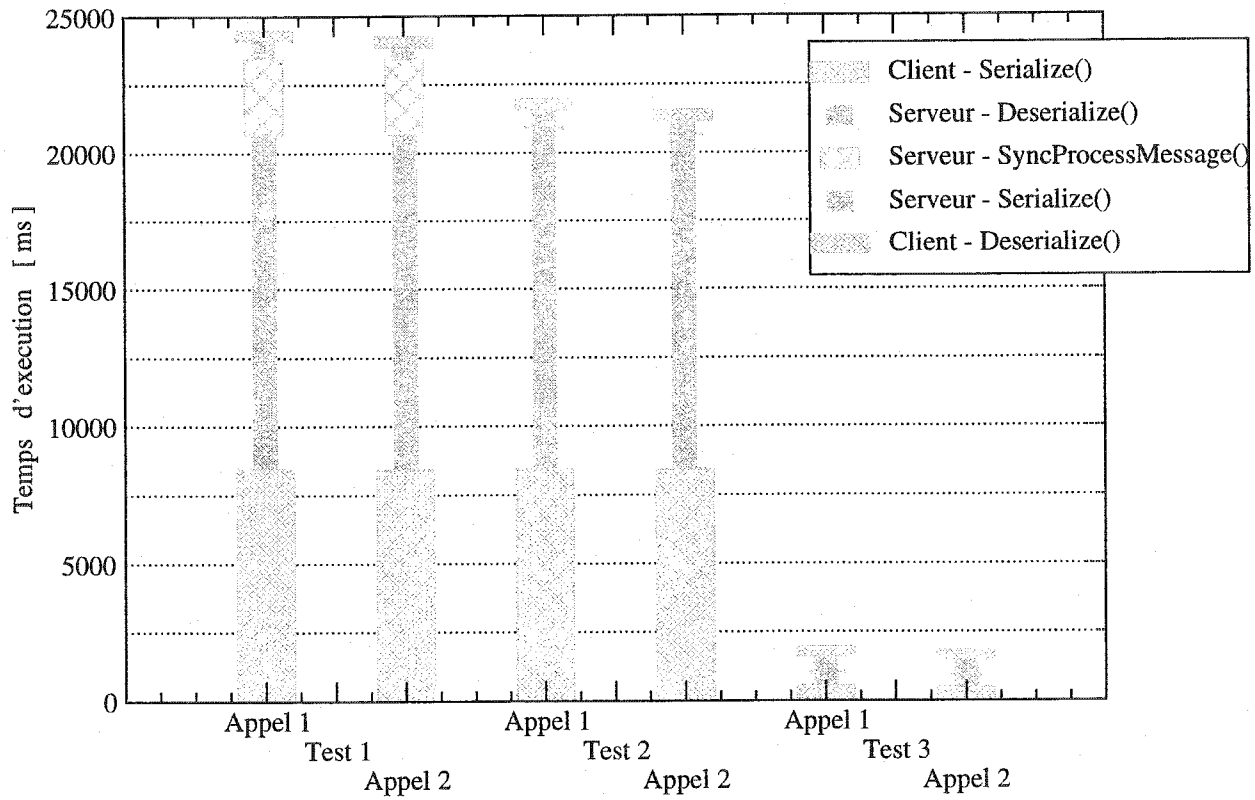

Figure 5.14 Les appels locaux - Tests 1,2,3

En utilisant l'outil proposé, les événements de chaque côté sont analysés simultanément et avec le même niveau de détail. Toute l'analyse est faite à l'exécution là où les goulots d'étranglement peuvent vraiment être observés. Une fois les goulots d'étranglement détectés par l'analyse de performance, l'optimisation des sections les plus lentes peut être entreprise.

L'amélioration de la performance peut se faire en modifiant l'application ou en changeant certaines classes par défaut de l'infrastructure. En effet, le *Remoting* offre une grande latitude à plusieurs niveaux (récepteurs de message, sérialisateur, canal).

Pour chaque application répartie, il faut qu'une analyse des données transférés entre les composants existe et il faut trouver le bon découpage de l'application (l'utilisation du cache, format et nombre des arguments, plusieurs petits appels versus quelques gros appels, envoi synchrone ou asynchrone).

La version actuelle de la distribution Mono (0.29) est fonctionnelle, mais peu optimisée. Ceci évoluera certainement avec la version 1.0 prévue pour le premier trimestre 2004. L'outil de performance présenté serait très utile, tant pour les utilisateurs que pour les développeurs du projet Mono.

CHAPITRE 6

CONCLUSION

Le protocole *.Net Remoting* représente une solution flexible pour le développement d'applications réparties. L'infrastructure contient des composants facilement configurables à plusieurs niveaux, qui assurent une transparence maximale. Ainsi, l'utilisation des services d'un objet distant ne demande pas plus d'effort que l'utilisation d'un objet local.

Derrière la facilité de développement d'applications réparties offerte par *.Net Remoting*, se cachent de nombreux sous-composants qui doivent être *configurés* et *coordonnés* dans un contexte d'exécution. L'analyse de performance de tels systèmes s'avère un défi qui demande des outils et des méthodes innovatrices, possiblement en adaptant ou améliorant les techniques classiques pour ce contexte.

L'outil présenté dans ce mémoire constitue une nouveauté pour l'analyse du comportement interne du CLI. Il fournit des renseignements qui n'étaient autrement pas disponibles. Il permet de mesurer avec grande precision la performance globale (le temps de réponse) d'une application répartie. Cette évaluation est fournie par la mesure des métriques de performance construites dans le contexte d'exécution (pour la sérialisation, désérialisation et l'exécution réelle de la méthode). À partir de la trace obtenue, il est possible de générer et de visualiser n'importe quel aspect du dialogue client–serveur par un remplacement ou un ajout de filtre.

Mono permet déjà de lire et d'exécuter du code Java et peut aussi dialoguer avec des applications CORBA [26], il est d'autant plus utile de mettre en valeur et d'augmenter les fonctionalités de l'outil d'analyse proposé dans ce mémoire.

Premièrement, il serait intéressant d'analyser le comportement de chaque fil d'exécution séparément (soit générique, soit appartenant à la réserve de fils d'exécution) pour des applications complexes. Une instrumentation sélective, au niveau

d'un module, d'un assemblage ou d'une seule méthode minimiserait l'impact sur l'application, permettant de trouver plus vite la cause de goulots d'étranglements dans le secteur sélectionné. Plusieurs analyses sont possibles à partir du nouveau contexte introduit par CLI: mesurer le temps passé pour le JIT (le coût de la compilation de chaque méthode cible), mesurer le temps nécessaire pour chaque phase du cycle de vie d'un récepteur Remoting (la formation, le couplage avec la chaîne entière, le traitement de chaque message). C'est le faible couplage entre les composants qui assure l'accès à toutes ces informations, au travers des métadonnées.

Dans la même ligne, un autre objectif de travaux futurs serait l'étude de la pénalité de performance apportées par les interrogations des métadonnées dans un appel *.Net Remoting* (la création des objets mandataires transparents, la signature de la méthode utilisée pour générer le message, la conversion du message en flot de données). Le protocole *.Net Remoting* permet de travailler avec des objets avec états. Cet aspect lui permet de servir de base aux prochaines applications réparties.

RÉFÉRENCES

[1] Technical report. WWW 20 Oct, 2003 http://www.atl.external.lmco.com/projects/QoS.

[2] Technical report. WWW 26 Nov, 2003 http://www.ingorammer.com.

[3] Technical report. NEWS 26 Nov, 2003 news://microsoft.public.dotnet.framework.remoting.

[4] Technical report. WWW 26 Nov, 2003 http://iiop-net.sourceforge.net.

[5] *The Ace ORB.* WWW 26 Nov, 2003 http://www.cs.wustl.edu/~schmidt/TAO.html.

[6] *Active Objects, Semantics, Internet and Security.* WWW 26 Nov, 2003 http://www-sop.inria.fr/oasis.

[7] *Borland Software Corporation.* WWW 26 Nov, 2003 http://www.borland.com/optimizeit.

[8] *Common Data Representation.* WWW. 26 Nov, 2003 http://www.omg.org/news/whitepapers/iiop.htm.

[9] *Globus Toolkit.* WWW. 26 Nov, 2003 http://www-unix.globus.org/toolkit.

[10] *gprof.* WWW 26 Nov, 2003 http://www.gnu.org/manual/gprof-2.9.1/gprof.html.

[11] *IBM Jinsight.* WWW 26 Nov, 2003 http://www.research.ibm.com/jinsight.

[12] *Institut de Recherche en Informatique et Systèmes Aléatoires.* WWW 26 Nov, 2003 http://www.irisa.fr.

[13] *Institut National de Recherche en Informatique et en Automatique.* WWW 26 Nov, 2003 http://www.inria.fr.

[14] *Interactive Control and Debugging of Distribution.* WWW 26 Nov, 2003 http://www-sop.inria.fr/oasis/ProActive/IC2D.

[15] *International Organization for Standardization.* WWW 26 Nov, 2003 http://www.iso.ch.

[16] *The Internet Communications Engine.* WWW 26 Nov, 2003 http://www.zeroc.com/ice.html.

[17] *IONA Orbix.* WWW 26 Nov, 2003 http://www.iona.com.

[18] *The Jakarta Project.* WWW 26 Nov, 2003 http://jakarta.apache.org/log4j.

[19] *Java profile browser.* WWW 26 Nov, 2003 http://www.physics.orst.edu/~bulatov/HyperProf.

[20] *A Java Profiling and Visualization Tool.* WWW 26 Nov, 2003 http://www.cs.umn.edu/Research/JaViz.

[21] *.NET Framework.* WWW 26 Nov, 2003 http://www.microsoft.com/net.

[22] *Network Time Protocol.* WWW 26 Nov, 2003 http://www.ntp.org.

[23] *Object Management Group.* WWW 26 Nov, 2003 http://www.omg.com.

[24] *omniORB.* WWW 26 Nov, 2003 http://omniorg.sourceforge.net.

[25] *Open Source Middleware.* WWW. 26 Nov, 2003 http://www.objectweb.org.

[26] *OpenLink Virtuoso Universal Server.* WWW 26 Nov, 2003 http://www.openlinksw.com/virtuoso.

[27] *OProfile.* WWW 26 Nov, 2003 http://oprofile.sourceforge.net.

[28] *ORBAcus.* WWW 26 Nov, 2003 http://www.orbacus.com.

[29] *ORBit.* WWW 26 Nov, 2003 http://orbit-resource.sourceforge.net.

[30] *Quest Software.* WWW 26 Nov, 2003 http://www.quest.com/jprobe.

[31] *Rational Rose - Quantify.* WWW. 26 Nov, 2003 http://www-306.ibm.com /software/rational.

[32] *VisiBroker.* WWW 26 Nov, 2003 http://info.borland.com/techpubs/visibroker.

[33] Dmitry Belikov. *Genuine Channel.* WWW 26 Nov, 2003 http://www. genuinechannels.com.

[34] R. Bladh and P. Arneng. Performance analysis of distributed object middleware technologies. Master's thesis, Blekinge Institute of Technology, Sweden, 2003.

[35] D. A. Carr and J. Moe. *Using Execution Trace Data to Improve Distributed Systems.* Software - Practice and Experience, 2002.

[36] Coulouris, Dollimore, and Kindberg. *Distributed Systems - Concepts and Design.* Addison-Wesley, 2001.

[37] Michel Dagenais. *Systèmes répartis sur l'Internet.* WWW 26 Nov, 2003 http://www.cours.polymtl.ca/inf4402.

[38] The Data Intensive Distributed Computing Research Group. *NetLogger Toolkit.* WWW 26 Nov, 2003 http://www-didc.lbl.gov/NetLogger.

[39] Alan Dennis. *.NET Multithreading.* Manning Publications Co., 2002.

[40] The Distributed System Research Group - University of Waterloo. *POET.* WWW 26 Nov, 2003 http://ccnga.uwaterloo.ca/poet.

[41] M. Litoiu et al. *A Performance Engineering Tool and Method for Distributing Applications.* IBM Centre for Advanced Studies Conference, 1997.

[42] P. Sandoz et al. *Fast Web Services.* WWW 26 Nov, 2003 http://developer.java.sun.com/developer/technicalArticles/WebServices/fastWS/index.html.

[43] Erich Gamma et all. *Design Patterns Book.* Addison-Wesley, 1995.

[44] H. Nielsen et all. *Direct Internet Message Encapsulation*, 2002. WWW 26 Nov, 2003 http://www-106.ibm.com/developerworks/webservices/library/ws-dime.

[45] Scott McLean et all. *.NET Remoting.* Microsoft Press, 2002.

[46] S. Fell and P.Drayton. *JABBER Channel.* WWW 26 Nov, 2003 http://www.ingorammer.com/Software/OpenSourceRemoting/OpenSourceMain.html.

[47] Intel. *Using the RDTSC Instruction for Performance Monitoring.* WWW 26 Nov, 2003 http://cedar.intel.com/software/idap/media/pdf/rdtscpm1.pdf.

[48] ECMA International. Partition I - Architecture. Technical report. WWW 26 Nov, 2003 http://www.ecma-international.org/publications/standards/Ecma-335.htm.

[49] ECMA International. Partition II - Metadata Definition and Semantics. Technical report. WWW 26 Nov, 2003 http://www.ecma-international.org/publications/standards/Ecma-335.htm.

[50] ECMA International. Partition III - CIL Instruction Set. Technical report. WWW 26 Nov, 2003 http://www.ecma-international.org/publications/standards/Ecma-335.htm.

[51] M. Juric. *Performance Comparison of CORBA and RMI.* WWW 26 Nov, 2003 http://lisa.uni-mb.si/~juric/paperss.htm.

[52] Roman Kiss. *DIME Channel.* WWW 26 Nov, 2003 http://www.codeproject.com/cs/webservices/remotingdime.asp.

[53] T. Kunz and M. F. H. Seuren. *Fast Detection of Communication Patterns in Distributed Executions.* IBM Centre for Advanced Studies Conference, 1997.

[54] L. Lamporte. *Time, Clocks and the Ordering of Events in a Distributed System.* Communications of the ACM, 1978.

[55] A. Mathew and M. Roulo. *Accelerate your RMI programming*, 2001. WWW. 26 Nov, 2003 http://www.javaworld.com/javaworld/jw-09-2001/jw-0907-rmi_p.html.

[56] Microsoft. Technical report. WWW 26 Nov, 2003 http://msdn.microsoft.com.

[57] Sun Microsystem. *RMI specifications.* WWW 26 Nov, 2003 http://java.sun.com/j2se/1.4.2/docs/guide/rmi.

[58] Sun Microsystem. *Remote Procedure Call, RFC 1057/1831*, 1995.

[59] Gary Nutt. *Communicating Across Application Domain.* WWW 26 Nov, 2003 http://www.nuvolan.com.

[60] Open Group CAE Specification. *Application Response Measurement.*

[61] Robert Orfali and Dan Harkey. *Client/Server Programming with Java and CORBA.* Wiley Computer Publishing, 1997.

[62] F. G. Ottogalli. *Observations et analyses quantitatives multi-niveaux d'applications à objets réparties.* PhD thesis, Institut d'informatique et Mathématiques Appliquées de Grenoble, 2001.

[63] Ingo Rammer. *SMTP Channel.* WWW 26 Nov, 2003 http://www.ingorammer.com/Software/OpenSourceRemoting/OpenSourceMain.html.

[64] Ingo Rammer. *Advanced .Net Remoting.* Apress, 2002.

[65] Jeffrey Richter. *Applied Microsoft.NET Framework Programming.* Microsoft, 2002.

[66] Lionel Senturier. *Middleware.* Laboratoire d'Informatique de Paris 6. WWW 26 Nov, 2003 http://www-src.lip6.fr/homepages/Lionel.Seinturier/middleware.

[67] Werner Vogels. *Web Services are not Distributed Objects - Common Misconceptions about Service Oriented Architectures.* WWW 26 Nov, 2003 http://weblogs.cs.cornell.edu/AllThingsDistributed/archives/000120.html.

[68] Ximian Inc. *Mono.* WWW 26 Nov, 2003 http://www.go-mono.com.

[69] Karim Yaghmour. *Linux Trace Toolkit.* WWW. 26 Nov, 2003 http://www.opersys.com/LTT.

ANNEXE I

Listing I.1 Les modifications de l'infrastructure Mono

```
#include <mono/metadata/profiler.h>
#include <mono/metadata/pedump.c>
#include <mono/metadata/metadata.h>
#include <asm/msr.h>
#include <glib.h>

static unsigned int tscH = 0;
static unsigned int tscL = 0;


/*
 * structure de données qui décrit la méthode et le moment d'exécution
 */
typedef struct _methodInOut {
        // valeurs possibles: 'E' ou 'L'
        char sens;

        // identificateur unique de la méthode
        unsigned int token;

        // le moins significatif octet du registre TSC
        unsigned int tscL;

        // le plus  significatif  octet du registre TSC
        unsigned int tscH;
}methodInOut;

/*
 * deux tampons pour garder les informations pendant l'exécution
 */
static methodInOut enter[128000]={0,0,0,0}, leave[128000]={0,0,0,0};

static unsigned int nbEnter=0;
static unsigned int nbLeave=0;


/*
 * exécute a chaque ENTER de l'appel
 */
static void
enter_method (MonoMethod *method, char *ebp)
{

        rdtsc(tscL, tscH);
        enter[nbEnter].sens ='E';
        enter[nbEnter].token = (unsigned int)method;
        enter[nbEnter].tscL =tscL;
        enter[nbEnter++].tscH =tscH;
```

```
        printf("\n%u#%s:%s",(unsigned int) method, method->klass->name, method->name);

        // exécuté au remplissage du tampon
        if (nbEnter >=128000)
        {
                write (8, enter, 128000 * sizeof (struct _methodInOut));
                nbEnter = 0;
        }

        //..... code Mono
}

/*
 * exécute a chaque LEAVE de l'appel
 */
static void
leave_method (MonoMethod *method, ...)
{

        char * classs;
        MonoImage *imageC;
        MonoMethodHeader *header;
        guint32 cols[MONO_TYPEDEF_SIZE];

        rdtsc(tscL, tscH);
        leave[nbLeave].sens = 'L';
        leave[nbLeave].token = (unsigned int)method;
        leave[nbLeave].tscL = tscL;
        leave[nbLeave++].tscH = tscH;

        // exécuté au remplissage du tampon
        if (nbLeave >=128000)
        {
                write (9, leave, 128000 * sizeof (struct _methodInOut));
                nbLeave = 0;
        }

        // exécuté au fin d'application
        if (method->name[0] == 'M' && method->token ==0 && (g_ascii_strcasecmp (method->name, "Main") == 0))
        {
                write (8, enter, nbEnter * sizeof (struct _methodInOut));
                write (9, leave, nbLeave * sizeof (struct _methodInOut));
        }

        //....code Mono
}
```

ANNEXE II

Listing II.1 L'implémentation MetInfo.dll

```
/*
 * Fichier: MetInfo.cs
 *
 * But: définir l'unité de base utilisé dans le traitement
 *              d'événements passés pendant l'exécution d'application
 *
 */

namespace MethodInfo {

using System;
using System.Collections;
using System.Diagnostics;
using System.IO;
using System.Text.RegularExpressions;


 ///<summary>
 /* classe qui décrit l'unité de base dans le traitement– l'événement */
 ///</summary>
 ///<remarks>
 /*
  * pour chaque méthode on doit avoir:
  * token, tokenParent, tEnter, tLeave
  */
 ///</remarks>

 public class MetInfo : IComparable {

        ///<summary>le temps cumulé</summary>
        public static double cumulativ;

        ///<summary>la frequénce de la machine</summary>
        public static double frequency;

        ///<summary>contient l'événement déclanché, qui est déja défini</summary>
        public string info;

        bool neverReturn;
        ///     <value>accès au membre 'neverReturn'</value>
        public bool NeverReturn {
                get {
                        return neverReturn;
                }
                set {
                        neverReturn = value;
                }
        }
```

```
///<summary>la valeur du registre TSC – le moins significatif octet</summary>
///<remarks>utilisé pour le temps d'entrée (tEnter)</remarks>
public UInt32 tscL;

///<summary>la valeur du registre TSC – le plus significatif octet </summary>
///<remarks>utilisé pour le temps d'entrée (tEnter)</remarks>
public UInt32 tscH;


///<summary>la valeur du registre TSC – le moins significatif octet </summary>
///<remarks>utilisé pour le temps de sortie (tLeave)</remarks>
public UInt32 tscLS;

///<summary>la valeur du registre TSC – le plus significatif octet </summary>
///<remarks>utilisé pour le temps de sortie (tLeave)</remarks>
public UInt32 tscHS;

///<summary>le temps propre d'exécution</summary>
public double self=0;

///<summary>le temps cumulé, si la méthode a des enfants</summary>
public double deltaCum = 0;

///<summary>le temps total d'exécution </summary>
///<remarks>delta = self + deltaCum </remarks>
///<remarks> deltaCum = Somme de 'delta' des enfants</remarks>
public double delta=0;

///<summary>le nom de la classe dont appartient la méthode</summary>
public string klass;

///<summary>le nom de la méthode</summary>
public string met;

///<summary>le nombre d'appels</summary>
///<remarks>non-utilisé dans cette version </remarks>
public int nbCalled = 0;

///<summary>la méthode parent </summary>
///<remarks> chaque méthode doit avoir un parent</remarks>
public MetInfo parent;

///<summary>le tableau avec les méthodes enfants sous appelées</summary>
public ArrayList childs;


///<value>l'identificateur unique de la méthode</value>
///<remarks></remarks>
private UInt32 token;

///<value>accès a l'identificateur de la méthode</value>
public UInt32 Token {
        get { return token; }
        set { token = value; }
}
```

```
///<summary>l'identificateur de la méthode parent</summary>
public UInt32 ParToken {
        get { return parent.token; }
        set { parent.token = value; }
}


///<summary>lit la fréquance via 'cat /proc/cpuinfo'</summary>
///<returns>la fréquance du processeur</returns>
///<param>void</param>
public static double GetFreq()
{
                ///<summary>la fréquance lue</summary>
                double f;
                string line;

                ///<summary>pairs clé-valeur</summary>
                string [] items = new string[2];

                Process freq = new Process();
                freq.StartInfo.FileName = @"/bin/cat";
                freq.StartInfo.Arguments = @"/proc/cpuinfo";
                freq.StartInfo.RedirectStandardOutput = true;
                freq.StartInfo.UseShellExecute = false;

                ///<summary>démarrage d'un processus</summary>
                freq.Start();
                while( (line=freq.StandardOutput.ReadLine()) != null)
                {
                        if(line.StartsWith("cpu MHz"))
                        {
                                items = line.Split(new Char[]{':'}, 2);
                                f = double.Parse(items[1]);
                                frequency = f;
                                return f;
                                break;
                        }
                }
                return 1;
        }


///<summary>méthode qui remplit le champ de nom - 'met'</summary>
///<remarks>appelée via délégué</remarks>
///<returns>void </returns>
///<param>nom du fichier dictionnaire</param>
///<param>l'identificateur cherché</param>
public void FillName(string dictionar, UInt32 id)
{
        string line;
        string[] hash = new string[2];
        FileStream dicFile;
        dicFile = new FileStream(dictionar, FileMode.Open, FileAccess.Read);
        StreamReader sr = new StreamReader(dicFile);
        line = sr.ReadLine();
```

```
        while((line=sr.ReadLine()) != null)
        {
                if(line.StartsWith(id.ToString()))
                {
                        hash = line.Split(new Char[]{'#'},2);
                        this.info = hash[1];
                        break;
                }
        }

        dicFile.Close();
}

///<summary>constructeur de la classe</summary>
public MetInfo(UInt32 tok, UInt32 highStart, UInt32 lowStart)
{
        // l'initialisation par défaut des champs
        parent =null;
        token = tok;
        tscL = lowStart;
        tscH = highStart;
        childs = new ArrayList();
        NeverReturn = false;
}

///<summary>constructeur </summary>
///<remarks>utilisé seulement quand LEAVE est rencontré</remarks>
public MetInfo(UInt32 tok)
{
        parent = null;
        token = tok;
        NeverReturn = false;
}

///<summary> la comparaison entre l'identificateur de la methode et l'objet courant 'this' </summary>
///<remarks>pour les comparaisons entre les identificateurs</remarks>
///<param> l'objet MetInfo de comparer</param>
///<returns> vrai, si les identificateurs ont la même valeur</returns>
public override bool Equals(object o)
{
        MetInfo m = (MetInfo) o;
        if (o == null)
                return false;

        if(this.Token.Equals(m.Token))
                return true;
        else
                return false;
}


///<summary>La classe est dérivée de l'interface IComparable</summary>
public int CompareTo (object obj)
{
        MetInfo x = obj as MetInfo;
        return Token.CompareTo(x.Token);
}
```

```
///<summary>ajoute une méthode enfant a la liste d'enfants du parent</summary>
///<returns>void </returns>
///<param>la méthode enfant trouvée</param>
public void AddChildren(MetInfo m)
{
        childs.Add(m);
}

///<summary>méthode qui complète les données de la méthode</summary>
///<remarks>appelée quand LEAVE est rencontré</remarks>
///<returns>void </returns>
///<param>les valeurs du registre TSC</param>
public void CompleteInit (uint highe, uint lowe)
{
        ulong diff=0;

        this.tscHS = highe;
        this.tscLS = lowe;

        diff = (((ulong)this.tscHS - this.tscH)<<32) + this.tscLS - this.tscL;

        this.delta = Convert.ToDouble(diff)/frequency/1000;

}

///<summary>l'affichage ds informations de la méthode </summary>
///<returns> void </returns>
///<param>la profondeur de l'enfant envers le plus ancêtre parent </param>
/*
public void Afficher(int x)
{

        ulong tEnter;

        tEnter = ((ulong)this.tscH<<32) + this.tscL;
        Console.WriteLine("{0}\t{1}\t{2}\t{3:f3}\t{4:f3}\t{5:f3}\t{6}\t{7}",this.Token,
                x, (40-x), this.delta, this.deltaCum, this.self, this.childs.Count, tEnter);
        if(this.childs.Count > 0)
        {
                x++;
                IEnumerator methodEnfant = this.childs.GetEnumerator();
                while(methodEnfant.MoveNext())
                {
                        MetInfo tempMethod = (MetInfo) methodEnfant.Current;
                        tempMethod.Afficher(x);
                }
        }
}
*/

///<summary>l'affichage ds informations de la méthode </summary>
///<returns> void </returns>
///<param>la profondeur de l'enfant envers le plus ancêtre parent </param>
public void AfficherVer2(int x)
{
```

```
                ulong tEnter;

                tEnter = ((ulong)this.tscH<<32) + this.tscL;
                cumulativ = cumulativ + this.self;
                Console.WriteLine("{0}\t{1}\t{2}\t{3:f3}\t{4:f3}\t{5:f3}\t{6}\t{7:f3}\t{8}\t{9:f3}",
                                this.Token,                     //0
                                x,                              //1
                                (40-x),                         //2
                                this.delta,                     //3
                                this.deltaCum,          //4
                                this.self,              //5
                                this.childs.Count,      //6
                                //tEnter,
                                cumulativ,              //7
                                this.info,              //8
                                tEnter/frequency/1000 //9
                                );
                if(this.childs.Count > 0)
                {
                        x++;
                        IEnumerator methodEnfant = this.childs.GetEnumerator();
                        while(methodEnfant.MoveNext())
                        {
                                MetInfo tempMethod = (MetInfo) methodEnfant.Current;
                                tempMethod.AfficherVer2(x);
                        }
                }
        }
 }
}
```

Listing II.2 L'implémentation du programme de traitement du côté serveur

```
/*
 * Fichier: RServer.cs
 *
 * But: chaque traitement d'un appel .Net Remoting comprend 4 événements
 *              abstraits du côté serveur
 *
 */

namespace ServerProcessCall {

using System;
using System.Collections;
using System.Text;
using System.Text.RegularExpressions;
using System.IO;

using MethodInfo;


        ///<summary>
        /* la classe de base pour les 4 événemnts abstraits */
        ///</summary>
        public class RServer
        {
                ///<summary>identificateur d'événement</summary>
                public string infoo;

                ///<summary>l'objet MetInfo correspondant</summary>
                public MetInfo method;

                ///<summary>la concaténation du registre TSC (MSB + LSB)</summary>
                public ulong time;

                ///<summary>le constructeur</summary>
                public RServer(MetInfo m)
                {
                        method = m;
                }

        }



        ///<summary>
        /* la classe qui décrit un appel complet de côté serveur */
        ///</summary>
        ///<remarks>
        /*conteneur pour les quatre instances de RServer*/
        ///</remarks>
        public class ProcessCalll
        {
        ///<summary>booléene qui détermine la fin de l'appel</summary>
                public bool completeProcessCalll;

        ///<summary>quatre instances RServer qui composent un objet ProcessCall</summary>
```

```
        public RServer serRcvRequest;
        public RServer serInvokeStart;
        public RServer serInvokeFinished;
        public RServer serSendReply;

    ///<summary>pendant l'appel peuvent s'élever des exceptions des méthodes</summary>
        public ArrayList exceptions;

        ///<summary>constructeur</summary>
        public ProcessCalll(RServer start)
        {
            completeProcessCalll = false;
            serRcvRequest = start;
            exceptions = new ArrayList(10);
        }

        ///<summary>constructeur</summary>
        public ProcessCalll()
        {
            completeProcessCalll = false;
            exceptions = new ArrayList(10);
        }


    }

///<summary>délégué qui démarre le processus GetFreq</summary>
    delegate double DelegateGetFreq();

///<summary>délégué qui remplit le nom de la méthode</summary>
    delegate void DelegateFillName(string d, UInt32 tok);


///<summary>
    /* la classe qui contient le point d'entrée principale- Main() */
    ///</summary>
    ///<remarks>traitement du fichier journal de côté serveur</remarks>
    public class TestApp
    {

        ///<summary>la traduction des événements abstraits</summary>
        public const string strTok1 = "InternalProcessMessage";
        public const string strTok2 = "RemotingServices:InternalExecuteMessage";
        public const string strTok3 = "ReturnMessage:.ctor";
        public const string strTok4 = "TcpMessageIO:SendMessageStream";

        static int count = 0;

        ///<summary>les variables statiques allouées pour les 4 événements</summary>
        static UInt32 token1;
        static UInt32 token2;
        static UInt32 token3;
        static UInt32 token4;

        ///<summary>la fréquance de la machine</summary>
        static double freq;
```

```
///<summary>permet l'accès a l'identificateur du premiere événement</summary>
public static UInt32 Token1 {
        get { return token1; }
        set { token1 = value; }
}

///<summary>permet l'accès a l'identificateur du deuxième événement</summary>
public static UInt32 Token2 {
        get { return token2; }
        set { token2 = value; }
}

///<summary>permet l'accès a l'identificateur du troisième événement</summary>
public static UInt32 Token3 {
        get { return token3; }
        set { token3 = value; }
}

///<summary>permet l'accès a l'identificateur du quatrième événement</summary>
public static UInt32 Token4 {
        get { return token4; }
        set { token4 = value; }
}


static string processCallsNumber;
///<summary>permet l'accès au champ 'processCallsNumber'</summary>
public static string ProcessCallsNumber {
        get { return processCallsNumber; }
        set { processCallsNumber = value; }
}


///<summary>le nombre d'appels .Net Remoting effectués</summary>
///<remarks> l'occurence de l'événement RS1</remarks>
static int nProcessCalls;
///<summary>permet l'accès au champ 'nProcessCalls</summary>
static int NProcessCalls {
        set { nProcessCalls = value; }
        get { return nProcessCalls; }
}



///<summary>le point d'entrée - Main() </summary>
///<returns> void </returns>
///<param>les deux fichiers: log.txt et dictionnaire.txt </param>

public static void Main(string [] args)
{
        if(args.Length != 2)
        {
                Console.WriteLine("\n\t Usage: prog.exe logFILE.txt dictionnare.txt\n");
                return;
        }

//      DelegateGetFreq dFreq = new DelegateGetFreq(MetInfo.GetFreq);
```

```
//      freq = dFreq();

        string line;

        ///<summary> </summary>
        string[] hash = new string[2];
        ///<summary>le deuxième argument: le dictionnaire</summary>
        string dFile = args[1];



        ///<summary>le flux alloué au fichier dictionnaire</summary>
        FileStream dicFile;



        try {
                dicFile = new FileStream(dFile, FileMode.Open, FileAccess.Read);
        }
        catch(Exception ex)
        {
                Console.WriteLine("\n\t error on file IO\n:{0}", ex.Message);
                return;
        }

        StreamReader sr = new StreamReader(dicFile);

        line = sr.ReadLine();

        ///<summary>l'identification des 4 événements dans le dictionnaire</summary>
        while(line != null  || ( count==3))
        {

                if(line.EndsWith(strTok1))
                {
                        count++;
                        hash = line.Split(new Char[]{'#'},2);
                        Token1 = UInt32.Parse(hash[0]);

                }

                if(line.EndsWith(strTok2))
                {
                        count++;
                        hash = line.Split(new Char[]{'#'},2);
                        Token2 = UInt32.Parse(hash[0]);
                        ProcessCallsNumber = "E " + Token2.ToString();
                }

                if(line.EndsWith(strTok3))
                {
                        count++;
                        hash = line.Split(new Char[]{'#'},2);
                        Token3 = UInt32.Parse(hash[0]);
                }

                if(line.EndsWith(strTok4))
```

```
        {
                count++;
                hash = line.Split(new Char[]{'#'},2);
                Token4 = UInt32.Parse(hash[0]);
        }

        line = sr.ReadLine();
}

dicFile.Close();

///<summary>le flux alloué au fichier journal</summary>
FileStream logFile;

try {
        logFile = new FileStream(args[0],FileMode.Open, FileAccess.Read);
}
catch(Exception ex)
{
        Console.WriteLine("Error on log File..\n {0} ", ex.ToString());
        return;
}


///<summary>vérifications</summary>
if(token1 == 0 && token2 == 0 && token3 ==0 && token4 ==0)
{
        Console.WriteLine(" token 1-4 did NOT initialize...!!");
        return;
}


///<summary>compte les appels existantes dans le fichier journal</summary>
StreamReader srLog = new StreamReader(logFile);
//callsNumber = "E " + token1.ToString();

while( (line=srLog.ReadLine()) != null)
{
        if(line.StartsWith(ProcessCallsNumber))
                NProcessCalls++;


}///fin fichier log.txt

//      Console.WriteLine(" {0} calls .. is it true? ", NProcessCalls);
//      Console.ReadLine();

logFile.Close();


///<summary>tous les appels existantes sont gardés dans un vecteur</summary>
///<remarks>chaque appel = RS1 + RS2 + RS3 + RS4</remarks>
ArrayList calls = new ArrayList(NProcessCalls);

///<summary>la pile utilisée dans le traitement</summary>
Stack stiva = new Stack();
```

```
///<summary>une liste de méthodes identifiées</summary>
ArrayList mList = new ArrayList();

///<summary>une liste d'exceptions trouvées</summary>
ArrayList exceptList = new ArrayList();

///<summary>l'élément du sommet de la pile</summary>
MetInfo topStack;

///<summary>l'identificateur du champ 'topStack'</summary>
UInt32 topToken;

///<summary>délégué pour remplir le nom de la métehode</summary>
DelegateFillName delegName;



try {
        logFile = new FileStream(args[0],FileMode.Open, FileAccess.Read);
}
catch(Exception ex)
{
        Console.WriteLine("Error on log_File..\n {0} ", ex.ToString());
        return;
}
srLog = new StreamReader(logFile);

///<summary>les éléments de traitement</summary>
Regex reg = new Regex(@"\s+");
string[] readed = new string[3];

Regex debut = new Regex("^[E|L]");
Match lineOK;


///<summary>la fin d'un appel</summary>
String endProcessCall = "L " + Token1.ToString();
Console.WriteLine("#####waittt, I'm processing ..");



RServer rSer;

while((line=srLog.ReadLine())!= null)
{
        lineOK = debut.Match(line);
        if (!lineOK.Success)
        {
                line = sr.ReadLine();
                continue;
        }

        readed = reg.Split(line);

        ///<summary>le sens d'appel est 'E' = entrée dans la méthode</summary>
        if(readed[0] == "E")
```

```
{
        MetInfo objTemp = new MetInfo(UInt32.Parse(readed[1]), UInt32.Parse(readed[2]),
              UInt32.Parse(readed[3]));
        if(stiva.Count>0)
        {
                topStack = (MetInfo) stiva.Peek();
                objTemp.parent = new MetInfo(topStack.Token);
                topStack.AddChildren(objTemp);
        }
        stiva.Push(objTemp);
}


if(Convert.ToUInt32(readed[1]) != token1)
{
        continue;
}

///<summary>une instance de traitement d'appel</summary>
ProcessCalll c = new ProcessCalll();


do
{
        readed = reg.Split(line);

        ///<summary>le sens d'appel est 'E' = entrée dans la méthode</summary>
        if(readed[0] =="E")
        {
                MetInfo obj = new MetInfo(UInt32.Parse(readed[1]), UInt32.Parse(readed[2]),
                      UInt32.Parse(readed[3]));
                if(stiva.Count>0)
                {
                        topStack = (MetInfo) stiva.Peek();

                        obj.parent = new MetInfo(topStack.Token);

                        topStack.AddChildren(obj);
                }



                if(obj.Token == token1)
                {
                        obj.info = "ServeurRecoitMessage";
                        rSer = new RServer(obj);
                        rSer.infoo = "ServeurRecoitMessage";
                        rSer.time = ((ulong)obj.tscH<<32) + obj.tscL;
                        c.serRcvRequest = rSer;
                        calls.Add(c);

                }

                if(obj.Token == token2)
                {
                        obj.info = "DebutInvocationServeur";
```

```
                rSer = new RServer(obj);
                rSer.infoo = "DebutInvocationServeur";
                rSer.time = ((ulong)obj.tscH<<32) + obj.tscL;
                c.serInvokeStart = rSer;
        }


        if(obj.Token == token3)
        {
                obj.info = "FinInvocationServeur";
                rSer = new RServer(obj);
                rSer.infoo = "FinInvocationServeur";
                rSer.time = ((ulong)obj.tscH<<32) + obj.tscL;
                c.serInvokeFinished = rSer;
        }

        if(obj.Token == token4)
        {
                obj.info = "ServeurEnvoieReponse";
                rSer = new RServer(obj);
                rSer.infoo = "ServeurEnvoieReponse";
                rSer.time = ((ulong)obj.tscH<<32) + obj.tscL;
                c.serSendReply = rSer;
                c.completeProcessCalll = true;
        }

        stiva.Push(obj);
}


///<summary>le sens d'appel est 'L' = la sortie de la méthode</summary>
if(readed[0] == "L")
{
        if(c.exceptions.Count != 0)
        {
                IEnumerator exception = c.exceptions.GetEnumerator();

                while(exception.MoveNext())
                {
                        MetInfo g = (MetInfo)exception.Current;
                        if(UInt32.Parse(readed[1]) == g.Token)
                        {
                                g.NeverReturn = false;
                                g.tscHS = UInt32.Parse(readed[2]);
                                g.tscLS = UInt32.Parse(readed[3]);
                                g.delta = g.tscLS - g.tscL ;
                                if(g.childs.Count>0)
                                {
                                        IEnumerator enChildd = g.childs.GetEnumerator();
                                        while (enChildd.MoveNext())
                                        {
                                                MetInfo temp1 = (MetInfo) enChildd.Current;
                                                g.deltaCum += temp1.delta;
                                        }
                                        g.self = g.delta - g.deltaCum;
                                }
                                else
```

```
                                g.self = g.delta;

                            mList.Add(g);

                            c.exceptions.Remove(g);
                            break;
                        }
                    }
                }///fin de la recherche de méthode dans les exceptions

while(stiva.Count>0)
{
        topStack = (MetInfo) stiva.Peek();
        topToken = topStack.Token;

        // la correspondance est parfaite: 'E' ~ 'L'
        if(UInt32.Parse(readed[1]) == topToken)
        {
                topStack = (MetInfo) stiva.Pop();

                // complète la description de la méthode
                topStack.CompleteInit (UInt32.Parse(readed[2]), UInt32.Parse(readed[3]));

                // invocation du délégué pour remplir le nom de la méthode
                delegName = new DelegateFillName(topStack.FillName);
                delegName(dFile, topToken);


                /*
                 * calcules internes: le temps cumulé des enfants, le temps propre d'
                      exécution
                 */
                if(topStack.childs.Count > 0)
                {
                        IEnumerator enChild = topStack.childs.GetEnumerator();
                        while (enChild.MoveNext())
                        {
                                MetInfo temp1 = (MetInfo) enChild.Current;
                                topStack.deltaCum += temp1.delta;
                        }
                        topStack.self = topStack.delta - topStack.deltaCum;
                }
                /* méthode sans enfants */
                else
                {
                        topStack.self = topStack.delta;
                }
                mList.Add(topStack);
                break;
        }
        topStack = (MetInfo) stiva.Pop();
        topStack.NeverReturn = true;

        // l'appel du délégué pour remplir le nom de la méthode
        delegName = new DelegateFillName(topStack.FillName);
        delegName(dFile, topStack.Token);
```

```
                                c.exceptions.Add(topStack);
                            }
                        }
                        line=srLog.ReadLine();
                    }
                    while(!line.StartsWith(endProcessCall));

                    topStack = (MetInfo) stiva.Pop();
                    topToken = topStack.Token;

                    // complète la description de la méthode
                    topStack.CompleteInit (UInt32.Parse(readed[2]), UInt32.Parse(readed[3]));

                    delegName = new DelegateFillName(topStack.FillName);
                    delegName(dFile, topToken);


                    /*
                     * calcules internes: le temps cumulé des enfants, le temps propre d'exécution
                     */
                    if(topStack.childs.Count > 0)
                    {
                        IEnumerator enChild = topStack.childs.GetEnumerator();
                        while (enChild.MoveNext())
                        {
                            MetInfo temp1 = (MetInfo) enChild.Current;
                            topStack.deltaCum += temp1.delta;
                        }
                            topStack.self = topStack.delta - topStack.deltaCum;
                    }
                    else
                    {
                        topStack.self = topStack.delta;
                    }
                    mList.Add(topStack);
                    line = srLog.ReadLine();
                }

                /*
                 * commence l'étape d'analyse de tous les appels lus
                 */
                Analyze(calls);


                logFile.Close();

            /*
                Console.WriteLine("\n In this log file I found : {0} calls", calls.Count);
                Console.WriteLine("\n In this log file I found : {0} exception ", exceptList.Count);
                IEnumerator enumException = exceptList.GetEnumerator();
                while(enumException.MoveNext())
                {
                    MetInfo methodTemp = (MetInfo)enumException.Current;
                    Console.Write(" # {0}", methodTemp.Token);
                }
            */
                // la fin du Main
```

```
}

///<summary>méthode d'analyse de tous les appels</summary>
///<returns>void</returns>
///<param>tableau d'appels</param>
public static void Analyze(ArrayList array)
{
        Console.WriteLine("\n\tProcessCall instances : {0}", array.Count);
        int j = 1;
        if(array.Count == 0)
        {
            Console.WriteLine("\n\t Erreure:  0 elements");
            return;
        }

        /*
         * pour chaque appel, commence la visualisation
         */
        IEnumerator enumAppel = array.GetEnumerator();
        while(enumAppel.MoveNext())
        {
            Console.WriteLine("\n\n");
            ProcessCalll tempProcessCall = (ProcessCalll) enumAppel.Current;
            Show(tempProcessCall,j);
            j++;
        }
}


///<summary>méthode de visualisation pour un cycle .Net Remoting</summary>
///<returns>void</returns>
///<param>objet 'ProcessCall'</param>
///<param>entier utilisé dans l'indentation</param>
public static void Show(ProcessCalll appel, int idx)
{
        int  offset =0;
        double t1, t2, t3;

        Console.WriteLine("#CALL-{0} completeProcessCall=[{1}] / [{2}] exceptions",
                                                idx, appel.completeProcessCalll, appel.exceptions.Count);

        if(appel.exceptions.Count > 0)
        {
            IEnumerator exceptEn = appel.exceptions.GetEnumerator();
            MetInfo t;
            while(exceptEn.MoveNext())
            {
                t = (MetInfo) exceptEn.Current;
                Console.WriteLine("#{0} / {1} ", t.Token, t.info);
            }
        }

        t1 = Convert.ToDouble(appel.serInvokeStart.time - appel.serRcvRequest.time) / freq /1000;
        t2 = Convert.ToDouble(appel.serInvokeFinished.time - appel.serRcvRequest.time)/freq/1000;
        t3 = Convert.ToDouble(appel.serSendReply.time - appel.serRcvRequest.time) / freq / 1000;
```

```
                    Console.WriteLine("#\t From client --->0--{0:f3}---{1:f3}---{2:f3}--->To client", t1, t2, t3)
                        ;

                    MetInfo tempus = (MetInfo) appel.serRcvRequest.method;
                    tempus.AfficherVer2(offset++);
        }

}


///<summary>classe pour faciliter le tri</summary>
///<remarks>tri après l'identificateur et le moment d'entrée</remarks>
public class SortByTokenAndEnter : IComparer
{
        public SortByTokenAndEnter()
        {}

        public int Compare (object o1, object o2)
        {
                MetInfo x = o1 as MetInfo;
                MetInfo y = o2 as MetInfo;


                //cas 1: le même identificateur, le moment d'entrée différent
                if (x.Token == y.Token)
                {
                        if (x.tscH == y.tscHS)
                                return x.tscL.CompareTo(y.tscL);
                        else
                                return x.tscH.CompareTo(y.tscH);

                }
                else
                        return x.Token.CompareTo(y.Token);

                //cas 2: le même identificateur, le même moment d'entrée: IMPOSSIBLE !

                //caz 3: les identificateurs differents, le même moment d'entrée

                /*
                if (x.Token < y.Token)
                        return y.Token.CompareTo(x.Token);
                else
                        return y.Token.CompareTo(x.Token);
                */
        }

}

///<summary>classe pour faciliter le tri</summary>
///<remarks>tri après le moment d'entrée</remarks>
public class SortByEnter1 : IComparer
{
        public SortByEnter1 ()
        {
        }
```

```
        public int Compare (object o1, object o2)
        {
                MetInfo x = o1 as MetInfo;
                MetInfo y = o2 as MetInfo;

                ulong timeM1 = ((ulong) x.tscH<<32) + x.tscL;
                ulong timeM2 = ((ulong) y.tscH<<32) + y.tscL;

                return timeM1.CompareTo(timeM2);
        }
}

}
```

ANNEXE III

Listing III.1 Exemple de script de visualisation

```
set key left
unset ytics
unset y2tics
#set terminal postscript landscape color
#set output "Appel.ps"
set xlabel "Appel␣.Net␣Remoting:␣execution␣Client␣[␣ms␣]"
set x2label "Appel␣.Net␣Remoting:␣execution␣Serveur␣[␣ms␣]"
set y2range [0:100] reverse
set yrange [0:100]
set mxtics 10
set mx2tics 10
set xtics mirror
set x2tics mirror
#
# appel 1
plot \
"<awk␣-f␣fClient.txt␣c.txt" i 0 axes x1y1 notitle w boxes, \
"<awk␣-f␣fServer.txt␣s.txt" i 0 axes x2y2 notitle w boxes, \
"<awk␣-f␣ssend.txt␣c.txt" i 0 axes x1y1 t "Client-Send" w points pt 9, \
"<awk␣-f␣srecv.txt␣c.txt" i 0 axes x1y1 t "Client-Receive" w points pt 11, \
"<awk␣-f␣ssend.txt␣s.txt" i 0 axes x2y2 t "Serveur-Send" w points pt 11, \
"<awk␣-f␣srecv.txt␣s.txt" i 0 axes x2y2 t "Serveur-Receive" w points pt 9
pause -1 "hit␣Enter␣for␣Call␣2␣"
#
# appel 2
plot \
"<awk␣-f␣fClient.txt␣c.txt" i 1 axes x1y1 notitle w boxes, \
"<awk␣-f␣fServer.txt␣s.txt" i 1 axes x2y2 notitle w boxes, \
"<awk␣-f␣ssend.txt␣c.txt" i 1 axes x1y1 t "Client-Send" w points pt 9, \
"<awk␣-f␣srecv.txt␣c.txt" i 1 axes x1y1 t "Client-Receive" w points pt 11, \
"<awk␣-f␣ssend.txt␣s.txt" i 1 axes x2y2 t "Serveur-Send" w points pt 11, \
"<awk␣-f␣srecv.txt␣s.txt" i 1 axes x2y2 t "Serveur-Receive" w points pt 9
pause -1 "hit␣Enter␣for␣Call␣3"
# ..
```